APLICAŢII MPI PENTRU SISTEME DE CALCUL PARALEL

Îndrumător de laborator

APLICAŢII MPI PENTRU SISTEME DE CALCUL PARALEL

Îndrumător de laborator

George Alexandru Nemneş
Tudor Luca Mitran
Adela Nicolaev
Lucian Ion

Cuprins

Introducere **7**

1 Programare paralelă cu procese independente şi fire de execuţie **9**
1.1 Procese independente - `fork` . . . 10
1.1.1 Apelul de sistem `exec` . . . 14
1.2 Fire de execuţie - threads . . . 17
1.2.1 Crearea şi gestionarea tread-urilor . . . 17
1.2.2 Variabile `mutex` . . . 26
1.2.3 Variabile de condiţionare . . . 30
1.3 Procese independente vs. fire de execuţie . . . 36
1.4 Comunicare între procese: Pipes şi Sockets . . . 37
1.4.1 Pipes . . . 38
1.4.2 Sockets . . . 42

2 Interfaţa MPI – Elemente introductive **51**
2.1 Concepte de bază în calcul paralel . . . 51
2.1.1 Avantajele calculului paralel . . . 51
2.1.2 Iniţializarea şi finalizarea mediului MPI . . . 51
2.1.3 Rangul unui proces . . . 52
2.1.4 Comunicatorul `MPI_COMM_WORLD` . . . 52
2.1.5 Compilare şi rulare . . . 53
2.2 Program MPI minimal – Hello world! . . . 53

3 Operaţii de comunicare point-to-point **55**
3.1 Funcţiile de comunicare `MPI_Send()` şi `MPI_Recv()` . . . 55
3.2 Ordinea operaţiilor in MPI . . . 57
3.3 Rutine de comunicare de tip non-blocking . . . 59

3.4 Tipuri de date 62
3.5 Aplicaţii 64
3.5.1 Calculul unei integrale 64
3.5.1.1 Metoda dreptunghiului. 64
3.5.1.2 Metoda Monte-Carlo 68

4 Operaţii de comunicare colective 73
4.1 Operaţii de tip broadcast, scatter, gather 73
4.1.1 `MPI_Bcast` 73
4.1.2 `MPI_Scatter` 75
4.1.3 `MPI_Gather` 77
4.1.4 Număr diferit de elemente - `MPI_Scatterv`, `MPI_Gatherv` 79
4.2 Operaţii de reducere globală – `MPI_Reduce` 80
4.3 Multi-broadcast 82
4.4 Multi-accumulation 83
4.5 Total exchange 84

5 Comunicatori şi grupuri 85
5.1 Funcţii MPI pentru operaţii cu grupuri şi comunicatori 86

6 Operaţii I/O în MPI 93
6.1 Funcţii I/O definite de standardul MPI 2.0 93
6.2 Exemple de utilizare a funcţiilor MPI pentru operaţii I/O 98

7 Implementarea schemelor master-slave, client-server, task pool, producer-consumner 103
7.1 Master-Slave 103
7.2 Client-Server 104
7.3 Task pool 104
7.4 Producer-Consumer 105

Referinţe 107

Introducere

Îndrumătorul de laborator se adresează studenţilor care urmează cursul *Arhitectura sistemelor de calcul paralel*. Pe parcursul celor şapte capitole, lucrarea prezintă conceptele generale de programare paralelă şi aplicaţii de laborator.

În primul capitol sunt introduse noţiunile de proces şi fir de execuţie, modalitaţi de gestionare a acestora şi comunicarea prin *pipes* şi *sockets*. În capitolul 2 este prezentată interfaţa *Message Passing Interface* (MPI): rangul unui proces, comunicatori şi un prim exemplu de cod MPI. În continuare sunt detaliate operaţiile de comunicare însoţite de exemple, mai întâi cele de tip *point-to-point*, în capitolul 3, şi ulterior, în capitolul 4, operaţiile de comunicare colective. În capitolul 5 sunt prezentate funcţii MPI pentru operaţii cu grupuri şi comunicatori. Operaţiile de input-output în mod paralel sunt discutate in capitolul 6. Ultimul capitol prezintă scheme generale utilizate în programare paralelă.

Sunt incluse segmente de cod, ca aplicaţii specifice pentru noţiunile discutate, folosind limbajul de programare C.

Capitolul 1

Programare paralelă cu procese independente şi fire de execuţie

Există mai multe opţiuni tehnice pentru implementarea unei aplicaţii numerice în formă paralelă. Acestea sunt în general puternic dependente de sistemul de operare şi de structura fizică a sistemului de calcul şi a reţelei de comunicare. În cele ce urmează vor fi prezentate metode valabile pentru sisteme de operare care au la bază un nucleu (kernel) de tip Unix (mai exact Linux), dar, în principiu, toate acestea au un echivalent şi pe alte sisteme de operare. Exemplele sunt implementate în limbajul C, dar aceste metode nu sunt în general limitate de limbajul de programare.

Opţiunile disponibile în programarea paralelă se referă la modul prin care sunt lansate procesele şi la modalitatea prin care acestea pot comunica între ele. Se poate vorbi astfel de modalităţi simplificate de lansare şi gestionare a unui grup de procese, cum este folosirea unor scripturi de tip shell pentru lansarea unui număr de programe şi gestionarea resurselor pe care acestea le au la dispoziţie, chiar şi pe mai multe maşini de calcul conectate la o reţea. Totuşi, aceste metode au o aplicabilitate limitată şi din acest motiv nu pot fi considerate abordări practice de calcul paralel.

În secţiunile următoare vor fi prezentate anumite metode populare folosite în dezvoltarea aplicaţiilor paralele cum sunt cele de lansare a unor procese copil independente de procesul părinte (prin *fork* sau *clone*), lansarea unor procese care împart anumite resurse cu părintele (cunoscute ca thread-uri), comunicare locală (pe aceeaşi maşină) între procese şi/sau thread-uri folosind *pipe*-uri sau comunicarea prin reţea (care nu exclude neapărat comunicarea locală) cu ajutorul *socket*-urilor [1]. Pe lângă aceste metode, există şi aplicaţii complexe menite să automatizeze şi să simplifice

efortul de paralelizare, cum sunt cele de tip MPI (abreviere de la Message Passing Interface) [2–5]. Pe lângă diversele implementări MPI, există şi alternative specilizate care sunt menite să îmbunătăţească anumite puncte slabe prezente în protocoalele de tip *message passing* (unul important fiind lipsa de redundanţă la erori sau probleme de hardware) sau care sunt pur şi simplu optimizate pentru anumite tipuri de operaţii. Deoarece în mediul academic şi ştiinţific MPI este în continuare protocolul de bază pentru paralelizare, acesta va fi prezentat pe larg în capitolele următoare.

1.1 Procese independente - `fork`

După cum a fost deja menţionat, două metode populare pentru implementarea aplicaţiilor paralele sunt pornirea de procese noi independente (folosind de exemplu funcţia `fork()`) sau pornirea unor thread-uri (cu ajutorul librariei Pthreads), caz în care procesele copil care rămân legate în continuare şi împart anumite resurse cu procesul părinte.

În mediul Unix, pentru a crea un nou proces, independent de cele deja existente, se foloseşte apelul de sistem (system call) `fork()`, al cărui prototip este

```
pid_t fork(void);
```

şi care este definit în *unistd.h*. `Fork()` respectă standardul POSIX (Portable Operating System Interface) şi are rolul de a crea un nou proces prin clonarea procesului părinte. Noul proces are un PID (identificatorul unic de proces) propriu şi nu împarte memoria cu părintele iar contorul de program (program counter) al procesului copil indică comanda imediat următoare funcţiei `fork()`.

Funcţia `fork()` are particularitatea de a întoarce două valori, una pentru procesul părinte şi una pentru procesul copil. În caz de succes va întoarce valoarea PID-ului în procesul părinte şi valoarea 0 în procesul copil, iar în caz de eşec întoarce valoarea -1 în procesul părinte şi niciun proces copil nu este creat. Pentru mai multe detalii referitoare la această funcţie se poate consulta intrarea de manual cu ajutorul comenzii *man fork*.

Pe lângă funcţia `fork()`, se vor utiliza şi apelurile de sistem `wait()`, `waitpid()` sau `waitid()`. Acestea sunt folosite când este necesar ca procesul părinte să aştepte o modificare a proceselor copil şi au prototipurile

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);.
```

Acestea indică dacă procesele copil s-au finalizat sau au fost oprite sau repornite printr-un semnal. Diferenţa majoră dintre cele trei este că `wait()` este o comandă generală care aşteaptă un semnal de la orice proces copil (din acest motiv trebuie apelată separat pentru fiecare proces copil), `waitpid()` primeşte ca parametru explicit PID-ul procesului pe care îl vizează iar `waitid()` este o funcţie mai flexibilă care poate aştepta un proces, un grup de procese sau orice proces. În exemplele care vor urma va fi folosită comanda `wait()` datorită caracterului general pe care îl are. `Wait()` are ca parametru de intrare variabila status care poate fi inspectat cu anumite macro-uri pentru a afla informaţii despre starea proceselor copil. Pentru mai multe detalii despre cele trei funcţii se poate consulta manualul cu ajutorul comenzii de terminal *man wait*.

În cazul în care una din funcţiile de aşteptare nu sunt utilizate, procesele copil devin procese *zombie*. Deşi aparent inofensive, procesele *zombie* pot fi periculoase deoarece, dacă nu sunt aşteptate de procesul părinte, nu sunt finalizate corespunzător iar PID-ul şi intrarea din tabela de procese (*process table*) rămân rezervate şi nu pot fi reutilizate. În momentul în care procesul părinte ia sfârşit, PID-ul şi intrarea din tabela de procese vor fi însă eliberate, iar dacă procesul părinte finalizează înaintea celor de tip copil, acestea din urmă vor fi adoptate de procesul superior lor (de obicei *init*) care le asigura un apel `wait()` (se poate verifica acest efect prin întârzierea execuţiei procesului copil cu funcţia `sleep()`, astfel încât procesul părinte să termine înaintea lui, şi afişarea PID-ului părintelui cu funcţia `getppid()` - în cazul adoptării procesului copil de către *init*, `getppid()` va întoarce valoarea 1). Cum funcţia `wait()` este de tip blocking, şi uneori acest aspect nu este de dorit, acesteia îi poate fi indicat flag-ul `WNOHANG` pentru a da posibilitatea procesului care o apelează să nu se blocheze în ea (devine nonblocking).

Următorul exemplu de program foloseşte comanda `fork()` pentru a porni trei procese noi din procesul părinte şi forţează procesul părinte să aştepte ca acestea să se încheie cu ajutorul comenzii `wait()`.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include <sys/types.h>
```

```c
#include <sys/wait.h>

int main(void)
{
  int i, status;
  //PID-ul unui proces copil va fi zero iar valoarea '1'
  //este folosita pentru a indica procesul parinte.
  pid_t my_pid=1;

  //Bucla pentru initializarea a trei procese copil.
  for(i=0;i<3;i++)
  {
    //Procesul parinte va fi singurul cu variabila
    //my_pid nenula.
    if(my_pid!=0)
    {
      //In caz de succes functia fork() va intoarce
      //valoarea PID a proceslui nou creat (procesul
      //copil). Daca functia fork intoarce valoarea -1
      //acest lucru indica o eroare in aparuta la
      //pornirea a noului proces.
      if( (my_pid = fork()) < 0)
      {
        perror("Fork error\n");
        exit(1);
      }
    }
  }

  //Se vor afisa: valoarea curenta intoarsa de fork(),
  //PID-ul procesului curent cu ajutorul functiei
  //getpid() si PID-ul parintelui procesului curent
  //cu ajutorul functiei getppid().
  printf("my_pid=%i, getpid=%i, getppid=%i\n",
          my_pid, getpid(), getppid());

  //Procesul parinte va fi singurul cu variabila my_pid
  //nenula.
  if(my_pid!=0)
  {
    for(i=0;i<3;i++)
    {
```

```
      //Ca in cazul functiei fork(), in caz de succes
      //functia wait() va intoarce valoarea PID-ului
      //procesului nou creat sau -1 in caz de esec.
      if (wait(&status) < 0)
      perror("Wait error");
      _exit(1);
    }
  }

  return 0;
}
```

Mesajul afişat de acest program trebuie să fie similar cu:

my_pid=0, getpid=10452, getppid=10451
my_pid=0, getpid=10453, getppid=10451
my_pid=10454, getpid=10451, getppid=10390
my_pid=0, getpid=10454, getppid=10451

unde, evident, valorile PID-urilor vor fi probabil diferite. Este important de remarcat faptul că ordinea în care for fi executate procesele nu este stabilită deoarece acestea pot fi executate chiar simultan. Această lipsă de secvenţialitate trebuie luată în considerare în cazul în care cele două procese depind unul de altul (de exemplu, accesează acelaşi fişier) pentru a nu da naştere condiţiilor de cursă (*race conditions*).

Un contraexemplu la folosirea corectă a funcţiilor de aşteptare este programul următor, care va da naştere unui proces *zombie*. Pentru a vedea cum poate apărea un astfel de proces se poate rula programul următor concomitent cu monitorizarea proceselor aflate în desfăşurare (de exemplu, cu ajutorul comenzii "top").

```
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
  pid_t my_pid;

  if( (my_pid = fork()) < 0)
  {
    perror("Fork error\n");
    exit(1);
  }
```

```
  //Procesul copil v-a fi suspendat pentru 20 de secunde
  //in schimb ce procesul parinte se incheie imediat.
  if (my_pid > 0)
    sleep (20);

  return 0;
}
```

"Top" ar trebui să afişeze şi procesul *zombie*:

```
Tasks:  243 total, 1 running, 241 sleeping, 0 stopped, 1 zombie.
```

1.1.1 Apelul de sistem exec

Pe lângă clonarea unor procese, `fork()` poate fi utilizată împreună cu un apel de sistem din familia *exec*. Aceste funcţii sunt folosite pentru a înlocui procesul curent cu un nou proces distinct. Din procesul iniţial doar PID-ul rămâne acelaşi. Din această familie de funcţii fac parte:

```
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg, ...,
              char * const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
int execvpe( const char *file, char *const argv[],
              char *const envp[]);
```

care sunt definite în *unistd.h*. Toate aceste funcţii folosesc pe fundal apelul de sistem `execve()` şi nu întorc o valoare decât în caz de eroare (în acest caz valoarea întoarsă este -1). După cum se poate vedea din prototipurile funcţiilor, litera "l" prezentă în numele funcţiilor `execl()`, `execlp()`, `execle()` denotă faptul că trebuie indicaţi explicit parametrii funcţiei, în cazurile `execv()`, `execvp()`, `execvpe()` (litera "v") parametrii sunt indicaţi printr-un şir de tip char*, pentru `execle()` şi `execvpe()` (litera "e") se poate indica mediul de execuţie (*environment*), iar în cazurile

`execlp()`, `execvp()` şi `execvpe()` (litera "p") nu este necesară indicarea în mod explicit calea spre executabil deoarece acestea o găsesc automat. Detalii suplimentare despre funcţiile din familia *exec* se pot obţine cu ajutorul comenzii *man exec*.

În exemplul următor, procesul copil va fi înlocuit de un nou proces (în acest caz *ls*, care va lista conţinutul directorului curent).

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
  //Argumentele pentru exec.
  char* args[4] = { "/bin/ls", "-l", ".", NULL} ;
  pid_t my_pid;
  int status;

  //Pornirea procesului copil.
  my_pid = fork();

  //Conditie valabila in cazul procesului copil.
  if (my_pid == 0)
  {
    //Executa comanda indicata in lista de argumente args.
    execv( args[0], args);

    //Iar in cazul folosirii functiei execl():
    //execl("/bin/ls","/bin/ls", "-l", ".", NULL);

    //In momentul apelului exec, procesul curent este inlocuit
    //de noul proces. Daca ajunge in acest punct inseamna ca
    //a avut loc o eroare.
    perror("Execve error");
  }

  //Conditie valabila in cazul procesului parinte.
  else if (my_pid > 0)
  {
    if( (my_pid = wait(&status)) < 0)
    {
```

```
      perror("Wait error");
      exit(1);
    }
  }

  else
  {
    perror("Fork error");
    _exit(1);
  }

  return 0;
}
```

După cum se poate vedea în exemplul de mai sus, doar procesul părinte apelează funcţia `exit()`. Deoarece procesul copil moşteneşte o parte din datele procesului părinte (*open file descriptors, open message queue descriptors, open directory streams*), nu trebuie ca desfăşurarea sau finalizarea acestuia să o afecteze pe cea a procesului părinte. Din această cauză se va folosi apelul de sistem `_exit()` în loc de funcţia `exit()` (care foloseşte la bază tot apelul de sistem `_exit()` dar are în plus şi alte atribute).

În cazul utilizării funcţiei `fork()` trebuie avută însă grijă să nu apară un proces în lanţ, cunoscut ca *fork bomb*, care poate duce la prăbuşirea sistemului. Mai jos este dat drept exemplu un astfel de program.

ATENŢIE: acest program nu trebuie rulat, este doar un exemplu de tipul "aşa nu" !

```
#include <unistd.h>

int main(void)
{
  pid_t my_pid;

  while(1)
  {
    if( (my_pid = fork()) < 0)
    {
      perror("Fork error\n");
      _exit(1);
    }
  }
```

```
    return 0;
}
```

1.2 Fire de execuţie - threads

O altă metodă de a obţine programe în format paralel este cea a folosirii firelor de execuţie (threads) cu ajutorul librăriei Pthreads (POSIX threads). În comparaţie cu procesele copil pornite prin `fork()`, firele de execuţie nu sunt nişte procese cu adevărat independente şi împart anumite resurse cu procesul părinte, de exemplu memoria din stivă (*heap*). Din acest motiv trebuie să se ţină cont de ordinea finalizării proceselor: procesul părinte trebuie întotdeauna să se încheie după procesele copil (firele de execuţie). Detalii suplimentare pot fi aflate cu ajutorul comenzii *man pthreads*.

Subrutinele API-ului (abreviere de la Application Programming Interface) Pthreads se pot împărţi în mai multe grupe: gestionarea thread-urilor prin rutine de tipul creării, detaşării sau unirii; rutinele de sincronizare, cunoscute şi ca *mutex*-uri (abreviere de la *mutual exclusion*); variabilele de condiţionare care se adresează problemei de comunicare între thread-uri care împart un *mutex*; rutinele care sincronizează ştergerea/citirea.

1.2.1 Crearea şi gestionarea tread-urilor

Similar cu modul în care se crează procese cu `fork()`, şi în cazul thread-urilor va exista un fir de execuţie iniţial care le va crea pe cele ulterioare. Funcţia care creează noile thread-uri este `pthreads_create()`, al cărui prototip este:

```
int pthread_create( pthread_t *thread, const pthread_attr_t *attr,
                    void *(*start_routine)(void*), void *arg);
```

şi care este definită în *pthread.h*. Argumentele pe care le primeşte această funcţie sunt: `thread` - identificatorul unic, `attr` - diverse atribute ale thread-ului (acestea nu pot fi modificate decât până în momentul creării tread-ului, nu şi ulterior, în timpul rulării), `start_routine` - rutina executată de thread după ce este creat, `arg` - argumentul pe care îl primeşte rutina indicată în `start_routine` (se pot indica mai multe argumente cu ajutorul unei structuri).

Pe lângă rutina de creare, există şi cea de finalizare: `pthread_exit()`. Aceasta este diferită de funcţiile `exit()` sau `return` pentru că suspendă doar execuţia thread-ului curent, nu a întregului proces.

În principiu, există un număr maxim de thread-uri pe care le va accepta sistemul de operare. Acest număr depinde de fiecare implementare în parte şi poate fi aflat rulând comanda *ulimit -a | grep "max user process"* în terminal.

Exemplul următor arată cum se pot crea trei fire de execuţie; acesta se compilează prin comanda *gcc -pthread -o threads threads.c* .

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

//Rutina executata de toate thread-urile in afara de main.
void *thread_routine(void *threadid)
{
   printf("I am thread no. %i\n", *(int *)threadid);
   pthread_exit(NULL);
}

int main (void)
{
   int i;

   //Main va crea 3 thread-uri.
   int n_threads=3;

   //Identificatoarele unice ale thread-urilor vor fi
   //stocate intr-un vector.
   pthread_t threads[n_threads];

   int ptc;

   for(i=0; i<n_threads; i++)
   {
      printf("Main: creating thread no. %i\n", i);
      ptc = pthread_create(&threads[i], NULL,
                           thread_routine, (void *)&i);
```

```
        if (ptc!=0)
        {
            perror("Pthread create error");
            exit(1);
        }
    }

    pthread_exit(NULL);
}
```

Acest program va afişa ceva similar cu:

Main: creating thread 0
Main: creating thread 1
I am thread 0
Main: creating thread 2
I am thread 1
I am thread 2

După cum se poate vedea, ordinea în care execută diversele fire nu este bine determinată, acest lucru fiind o trăsătură a proceselor concurente. Însă în acest exemplu a fost inclusă o greşeală care nu este imediat evidentă. Dacă rutina de execuţie a thread-urilor este modificată adăugând o întârziere, problema devine imediat evidentă:

```
void *thread_routine(void *threadid)
{
    sleep(2);
    printf("I am thread %i\n", *(int *)threadid);
    pthread_exit(NULL);
}
```

caz în care ieşirea programului va afişa:

Main: creating thread no. 0
Main: creating thread no. 1
Main: creating thread no. 2
I am thread no. 3

I am thread no. 3
I am thread no. 3

ceea ce este evident greşit. Această problemă apare din cauză că rutina thread-urilor primeşte ca atribut adresa unei variabile care se modifică înainte să fie citită. Dacă nu ar fi fost utilizată funcţia de întârziere *sleep()*, această problemă ar fi trecut neobservată. Pentru a o rezolva se pot folosi cel putin două abordări: fie se pasează thread-urilor ca argument adresa unei variabile a cărei valori nu se schimbă

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

//Rutina executata de toate thread-urile in afara de main.
void *thread_routine(void *threadid)
{
   sleep(2);
   printf("I am thread %i\n", *(int *)threadid);
   pthread_exit(NULL);
}

int main (void)
{
   int i;

   //Main va crea 3 thread-uri.
   int n_threads=3;
   int x[n_threads];

   //Identificatoarele unice ale thread-urilor vor fi
   //stocate intr-un vector.
   pthread_t threads[n_threads];

   int ptc;

   for(i=0; i<n_threads; i++)
   {
      x[i]=i;
      printf("Main: creating thread %i\n", i);
      ptc = pthread_create(&threads[i], NULL,
```

```
                                thread_routine, (void *)&x[i]);

        if (ptc!=0)
        {
            perror("Pthread create error");
            exit(1);
        }
    }

    pthread_exit(NULL);
}
```

sau se foloseşte un artificiu, uneori periculos, de a avea o valoare de tip `long`, care are aceeaşi dimensiune cu un pointer de tip *void* `(void *)` - această abordare poate să dea erori în funcţie de arhitectura sistemului.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

//Rutina executata de toate thread-urile in afara de main.
void *thread_routine(void *threadid)
{
    sleep(2);
    printf("I am thread %li\n", (long)threadid);
    pthread_exit(NULL);
}

int main (void)
{
    long i;

    //Main va crea 3 thread-uri.
    int n_threads=3;

    //Identificatoarele unice ale thread-urilor vor fi
    //stocate intr-un vector.
    pthread_t threads[n_threads];

    int ptc;
```

```
    for(i=0; i<n_threads; i++)
    {
        printf("Main: creating thread %li\n", i);
        ptc = pthread_create(&threads[i], NULL,
                             thread_routine, (void *)i);

        if (ptc!=0)
        {
            perror("Pthread create error");
            exit(1);
        }
    }

    pthread_exit(NULL);
}
```

O funcţie utilă cu care se poate controla ordinea în care finalizează firele de execuţie este `pthread_join()`, cu prototipul:

```
int pthread_join( pthread_t thread, void **value_ptr);
```

Această funcţie suspendă execuţia thread-ului care o apelează până când firul de execuţie ţintă *thread* finalizează.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

//Rutina executata de toate thread-urile in afara de main.
void *thread_routine(void *threadid)
{
    sleep(2);
    printf("I am thread %i\n", *(int *)threadid);
    pthread_exit(NULL);
}

int main (void)
{
    int i;
```

```
void *status;

//Main va crea 3 thread-uri.
int n_threads=3;

int x[n_threads];
//Identificatoarele unice ale thread-urilor vor fi
//stocate intr-un vector.
pthread_t threads[n_threads];

//Variabila care stocheaza atributele thread-ului.
pthread_attr_t attr;

int ptc, ptj;

//Variabila atributelor este initializata.
pthread_attr_init(&attr);

//Este setat atributul 'joinable' care permite thread-ului
//unirea ulterioara prin functia pthread_join(). Acest
//lucru se face mai mult pentru siguranta deoarece
//tread-urile sunt in mod explicit 'joinable'.
pthread_attr_setdetachstate(&attr,
                            PTHREAD_CREATE_JOINABLE);

for(i=0; i<n_threads; i++)
{
   x[i]=i;
   printf("Main: creating thread %i\n", i);
   ptc = pthread_create(&threads[i], &attr,
                        thread_routine, (void *)&x[i]);

   if (ptc!=0)
   {
      perror("Pthread create error");
      exit(1);
   }
}

//Variabila atributelor este eliberata.
pthread_attr_destroy(&attr);
```

```
    for(i=0; i<n_threads; i++)
    {
        //Thread-ul curent (adica 'main()') astepta thread-urile
        //din sirul thread[t] sa se incheie.
        ptj = pthread_join(threads[i], &status);
        if (ptj!=0)
        {
            perror("Pthread join error\n");
            exit(1);
        }
    }

    pthread_exit(NULL);
}
```

Se poate observa în exemplul de mai sus folosirea atributului `PTHREAD_CREATE_JOINABLE` care indică în mod explicit că un fir de execuţie aşteaptă finalizarea altuia sau altora (este similar cu `wait()` pentru procese independente). Dacă thread-urile sunt *joinable* dar nu sunt aşteptate, acestea blochează resurse şi se comportă similar cu procesele *zombie*. În cazul în care thread-urile nu trebuie aşteptate, se poate folosi atributul `PTHREAD_CREATE_DETACHED`, astfel încât ID-ul şi resursele ocupate de firele de execuţie să poată fi reutilizate imediat.

Exemplul următor arată utilizarea unui alt atribut util al tread-urilor, cel de obţinere şi stabilire a dimensiunii stivei (stack-ului) cu funcţiile `pthread_attr_getstacksize()` şi `pthread_attr_setstacksize()` ale căror prototipuri sunt:

```
int pthread_attr_getstacksize( const pthread_attr_t *restrict attr,
              size_t *restrict stacksize);

int pthread_attr_setstacksize( pthread_attr_t *attr, size_t stacksize);
```

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
```

```c
//Atribute globale, variabile pentru toate thread-urile.
pthread_attr_t attr;

//Rutina executata de toate thread-urile in afara de main.
void *thread_routine(void *threadid)
{
   size_t mystacksize;
   pthread_attr_getstacksize (&attr, &mystacksize);
   printf("I am thread %i and my stack is %i bytes\n",
          *(int *)threadid, mystacksize);
   pthread_exit(NULL);
}

int main(void)
{
   int i;
   int ptc;
   size_t stack_size;

   //Main va crea 3 thread-uri.
   int n_threads=3;
   int x[n_threads];

   //Identificatoarele unice ale thread-urilor vor fi
   //stocate intr-un vector.
   pthread_t threads[n_threads];

   //Variabila atributelor este initializata.
   pthread_attr_init(&attr);

   //Se obtine dimensiunea curenta a stack-ului.
   pthread_attr_getstacksize (&attr, &stack_size);

   printf("Initial stack size = %li\n", stack_size);

   stack_size = 10000000;

   //Se stabileste noua dimensiune a stack-ului.
   pthread_attr_setstacksize (&attr, stack_size);

   for(i=0; i<n_threads; i++)
   {
```

```
        x[i]=i;
        printf("Main: creating thread %li\n", i);
        ptc = pthread_create(&threads[i], &attr,
                             thread_routine, (void *)&x[i]);

        if (ptc!=0)
        {
           perror("Pthread create error");
           exit(1);
        }
    }

    pthread_exit(NULL);
}
```

1.2.2 Variabile `mutex`

Variabilele `mutex` sunt principala modalitate prin care se poate implementa sincronizarea thread-urilor pentru a corela operaţiile multiple de scriere şi citire. `Mutex`-ul are rolul unei siguranţe care protejează accesul la resurse comune. În cazul librăriei Pthreads, `mutex`-ul se comportă ca o ştafetă prin care numai un thread poate deţine dreptul de scriere sau citire la un moment dat de timp. Celelalte thread-uri obţin drepturi de scriere/citire numai după ce variabila `mutex` este deblocată. `Mutex`-urile sunt utile pentru a preîntâmpina condiţiile de cursă care pot apărea şi în situaţii concrete precum rezervarea de locuri la cinema sau teatru (dacă un loc apare liber pe parcursul procesului de rezervare, acesta poate ajunge să fie cumpărat de mai multe persoane) sau tranzacţiilor financiare (dacă dintr-un cont sunt făcute simultan mai multe plăţi cu sume care individual depăşesc valoarea aflată în cont există riscul ca ambele să fie aprobate sau refuzate). Este obligatoriu ca variabilele globale considerate critice să fie modificate cu ajutorul unui mutex.

Rutinele necesare iniţializării şi distrugerii unui `mutex` sunt `pthread_mutex_init()` şi `pthread_mutex_destroy()` ale căror prototipuri sunt:

```
int pthread_mutex_init( pthread_mutex_t *restrict mutex,
              const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy( pthread_mutex_t *mutex);
```

Funcţia `pthread_mutex_init()` iniţializează `mutex`-ul cu atributele specificate de variabilă `attr` (în cazul `attr=NULL` `mutex`-ul este iniţializat cu valori implicite). O variabilă mutex iniţializată cu succes este din start deblocată. Funcţia `pthread_mutex_destroy()` distruge o variabilă `mutex` şi o readuce la starea de neiniţializare; această operaţie este reversibilă, iar `mutex`-ul poate fi reiniţializat prin `pthread_mutex_init()`. Orice utilizare a unui `mutex` distrus este nedefinită. Un `mutex` poate fi distrus în condiţii de siguranţă numai dacă este deblocat de toate thread-urile.

Funcţii cu efecte similare există şi pentru variabila de atribute:

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Blocarea şi deblocarea `mutex`-urilor se face cu ajutorul funcţiilor:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Rutina `pthread_mutex_lock()` este apelată de un thread pentru a bloca o anumită variabilă `mutex`. Dacă variabila a fost deja blocată de un alt thread, `pthread_mutex_lock()` va ţine ocupat thread-ul care o apelează până când variabila `mutex` este deblocată.

Pentru a nu ţine un fir de execuţie ocupat până la deblocarea `mutex`-ului, se poate utiliza funcţia `pthread_mutex_trylock()` care, în cazul unui `mutex` blocat, va întoarce imediat un cod de eroare şi va permite thread-ului să-şi continue activitatea. Această rutină este utilă în prevenirea interblocărilor (*deadlock*) prin care firele de execuţie ajung să se blocheze unele pe altele. Un exemplu tipic pentru deadlock ar fi cazul în care firul A blochează printr-un `mutex` o resursă X iar firul B blochează o resursă Y dar, înainte ca ambele să-şi elibereze `mutex`-urile, au nevoie de acces la resursa celuilalt fir (A la Y şi B la X).

Funcţia `pthread_mutex_unlock()` are rolul deblocării `mutex`-ului deţinut de

thread-ul care o apelează. Este obligatoriu ca această rutină să fie apelată după ce un thread nu mai are nevoie de acces la date protejate printr-un `mutex` pentru a le putea permite accesul celorlalte thread-uri. Funcţia va întoarce o eroare dacă `mutex`-ul a fost deja deblocat sau dacă este blocat de un alt thread.

Exemplul următor pune în aplicaţie o un `mutex` pentru a modifica o variabilă alocată în stiva funcţiei `main()`:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

typedef struct
{
  char *message;
  int id;
  int *val;
} my_struct;

pthread_mutex_t my_mutex;

//Rutina executata de toate thread-urile in afara de main.
void *thread_routine(void *thr_str)
{
   //Daca mutex-ul este deblocat, thread-ul preia controlul
   //asupra lui.
   pthread_mutex_lock (&my_mutex);

   printf("%s %i and my value is %i\n",
          (*(my_struct *)thr_str).message,
          (*(my_struct *)thr_str).id,
          *(*(my_struct *)thr_str).val);

   //Thread-ul incrementeaza valoarea variabilei x (definita
   //in functia main).
   (*(*(my_struct *)thr_str).val)++;

   //Thread-ul deblocheaza mutex-ul
   pthread_mutex_unlock (&my_mutex);
```

```
    pthread_exit(NULL);
}

int main (void)
{
    int i, x=0;

    //Main va crea 3 thread-uri.
    int n_threads=3;

    int ptc, ptj;
    pthread_attr_t attr;
    void *status;

    //Mesajul afisat de thread-uri.
    char my_message[]="I am thread";

    //Fiecare structura asociata unui thread este
    //stocata separat.
    my_struct thr_str[n_threads];

    //Identificatoarele unice ale thread-urilor vor fi
    //stocate intr-un vector.
    pthread_t threads[n_threads];

    //Initializarea variabilei mutex.
    pthread_mutex_init(&my_mutex, NULL);

    //Initializarea variabilei atributelor.
    pthread_attr_init(&attr);

    //Threadurile sunt de tip 'joinable'.
    pthread_attr_setdetachstate(&attr,
                                PTHREAD_CREATE_JOINABLE);

    for(i=0; i<n_threads; i++)
    {
        thr_str[i].message=my_message;
        thr_str[i].id=i;
        thr_str[i].val=&x;
        printf("Main: creating thread %i\n", i);
        ptc = pthread_create(&threads[i], &attr,
```

```
                    thread_routine, (void *)&thr_str[i]);

        if (ptc!=0)
        {
            perror("Pthread error");
            exit(1);
        }
    }

    for(i=0; i<n_threads; i++)
    {
        //Thread-ul curent (adica 'main()') asteapta
        //thread-urile din sirul threads[t] sa finalizeze.
        ptj = pthread_join(threads[i], &status);
        if (ptj!=0)
        {
            perror("Pthread join error\n");
            exit(1);
        }
    }

    //Variabila atributelor este eliberata.
    pthread_attr_destroy(&attr);

    //Variabila mutex este eliberata.
    pthread_mutex_destroy(&my_mutex);

    pthread_exit(NULL);
}
```

1.2.3 Variabile de condiţionare

Deşi variabilele `mutex` sunt folosite ca metode de blocare sau permitere unui thread să acceseze o resursă, acestea sunt uneori ineficiente din cauză că ţin thread-ul ocupat cu verificarea stării `mutex`-ului (până la eliberarea acestuia). Pentru a scăpa de această problemă se pot folosi, în paralel cu un `mutex`, *variabile de condiţionare* care înlocuiesc acest proces dinamic de verificare cu o funcţie de aşteptare.

Funcţiile necesare pentru gestionarea variabilelor de condiţionare sunt:

```
int pthread_cond_init(pthread_cond_t *restrict cond,
```

```
                const pthread_condattr_t *restrict attr);

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

.

`Pthread_cond_init()` are rolul iniţializării variabilei `cond` cu atributele `attr`. `Pthread_cond_destroy()` distruge variabila `cond` şi o readuce la o stare neiniţializată. Variabila de condiţionare poate fi reiniţializată după ce a fost distrusă. Folosirea unei variabile de condiţionare neiniţializată sau distrusă prin `pthread_cond_destroy()` duce la efecte nedefinite.

Funcţiile `pthread_condattr_init()` şi `pthread_condattr_destroy()` au rolul de a iniţializa/distruge atributele variabilei de condiţionare.

Funcţiile de control asupra thread-urilor care au la bază variabilele de condiţionare sunt:

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                pthread_mutex_t *restrict mutex,
                const struct timespec *restrict abstime);

int pthread_cond_wait(pthread_cond_t *restrict cond,
                pthread_mutex_t *restrict mutex);

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_signal(pthread_cond_t *cond);
```

`Pthread_cond_timedwait()` şi `pthread_cond_wait()` blochează execuţia unui thread folosind variabila de condiţionare `cond`. Cele două funcţii sunt echivalente, singura excepţie fiind că `pthread_cond_timedwait()` întoarce o eroare dacă timpul indicat de sistem a depăşit timpul absolut `abstime`. Ambele funcţii trebuie apelate cu variabila `mutex` blocată, în caz contrar fiind posibilă apariţia unor efecte nedefinite. Aceste funcţii deblochează în mod automat `mutex`-ul după ce sunt apelate şi blochează

în acelaşi timp thread-ul care le-a apelat. În cazul ambelor funcţii trebuie utilizat doar un `mutex` pentru operaţiile cu o anume variabilă de condiţionare.

Funcţiile `pthread_cond_signal()` şi `pthread_cond_broadcast()` sunt folosite pentru deblocarea a cel puţin un thread care a fost blocat prin variabila de codiţionare şi, respectiv, a tuturor thread-urilor blocate.

Exemplul următor arată modul în care se poate bloca activitatea unui thread (cel cu numărul 3) până când o anumită condiţie este împlinită (variabila incrementată depăşeşte valoarea 10).

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

//Valoarea pana la care va numara fiecare thread.
#define COUNT_UP_TO 5

//Valoarea de la care thread-ul 3 va incepe sa numere.
#define COUNT_LIMIT 5

//Valori definite global pentru ca sunt impartite de
//toate thread-urile si de functia main().
int count = 0;
pthread_mutex_t my_mutex;
pthread_cond_t my_condition_variable;

//Rutina de numarare simpla (fara asteptare) a
//thread-urilor 1 si 2.
void *just_count(void *id)
{
  int i;

  for (i=0; i < COUNT_UP_TO; i++)
  {
    //Se blocheaza mutex-ul.
    pthread_mutex_lock(&my_mutex);
    count++;

    //Verifica daca a ajuns la valoarea limita. Aceasta
    //verificare se face numai cu mutex-ul blocat pentru
    //ca valoara comparata sa nu poata fi modificata in
```

```
    //acest timp.
    if (count == COUNT_LIMIT)
    {
      printf("Thread %i has now reached the threshold\n",
             *(int *)id, count);
      pthread_cond_signal(&my_condition_variable);
      printf("Thread %i has send a signal to thread 3 "
              "to start\n",*(int *)id);
    }

    printf("Thread %i has counted up to %i\n",
           *(int *)id, count);

    //Se deblocheaza mutex-ul.
    pthread_mutex_unlock(&my_mutex);

    //Intarziere temporala pentru a permite si celorlalte
    //thread-uri sa fie activate.
    sleep(1);
   }
  pthread_exit(NULL);
}

//Rutina de numarare a tread-ului 3 (cu asteptare).
void *wait_and_count(void *id)
{
  int i;
  printf("Thread %i is now waiting \n", *(int *)id);

  //Mutex-ul este blocat pentru a putea apela functia
  //pthread_cond_wait().
  pthread_mutex_lock(&my_mutex);
  while (count < COUNT_LIMIT)
  {
    //Odata apelata, functia pthread_cond_wait() deblocheaza
    //in mod automat mutex-ul.
    pthread_cond_wait(&my_condition_variable, &my_mutex);
  }

  //In acest punct mutex-ul apartine din nou thread-ului 3
  //pentru ca pthread_cond_signal() l-a deblocat.
```

```
  printf("Thread %i is no longer waiting\n", *(int *)id);
  printf("Thread %i is now unlocking the mutex\n",
         *(int *)id);

  pthread_mutex_unlock(&my_mutex);

  for(i=0;i<COUNT_UP_TO;i++)
  {
    pthread_mutex_lock(&my_mutex);
    count++;
    printf("Thread %i has counted up to %i\n",
           *(int *)id, count);
    pthread_mutex_unlock(&my_mutex);
    sleep(1);
  }
  pthread_exit(NULL);
}

int main(void)
{
  int i, rc;

  //Main va crea 3 thread-uri.
  int n_threads=3;

  int x=1, y=2, z=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  //Se initializeaza mutex-ul si variabila de conditionare.
  pthread_mutex_init(&my_mutex, NULL);
  pthread_cond_init(&my_condition_variable, NULL);

  //Thread-urile primesc in mod explicit atributul 'joinable'.
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
                              PTHREAD_CREATE_JOINABLE);

  //Thread-urile sunt create, fiecare cu propria rutina
  //de executie.
  pthread_create(&threads[0], &attr,
                 just_count, (void *)&x);
```

```
  pthread_create(&threads[1], &attr,
                 just_count, (void *)&y);
  pthread_create(&threads[2], &attr,
                 wait_and_count, (void *)&z);

  //Asteapta thread-urile sa finalizeze.
  for (i = 0; i < n_threads; i++)
  {
    pthread_join(threads[i], NULL);
  }

  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&my_mutex);
  pthread_cond_destroy(&my_condition_variable);
  pthread_exit (NULL);
}
```

Iar un exemplu de output:

Thread 2 has counted up to 1
Thread 3 is now waiting
Thread 1 has counted up to 2
Thread 2 has counted up to 3
Thread 1 has counted up to 4
Thread 2 has now reached the threshold
Thread 2 has send a signal to thread 3 to start
Thread 2 has counted up to 5
Thread 3 is no longer waiting
Thread 3 is now unlocking the mutex
Thread 3 has counted up to 6
Thread 1 has counted up to 7
Thread 2 has counted up to 8
Thread 3 has counted up to 9
Thread 1 has counted up to 10
Thread 2 has counted up to 11
Thread 3 has counted up to 12
Thread 1 has counted up to 13
Thread 3 has counted up to 14
Thread 3 has counted up to 15

1.3 Procese independente vs. fire de execuţie

Un aspect important al programării paralele constă în alegerea abordării cele mai potrivite pentru o anume situaţie dată. Din acest motiv, este necesară compararea avantajelor folosirii proceselor independente sau a thread-urilor. În plus, în mediul Linux, `fork()` şi Pthreads sunt de fapt implementate pe baza aceluiaşi apel de sistem, `clone()`, care este definit în *sched.h*. Diferenţa de implementare este dată de parametrii diferiţi folosiţi în apelul `clone()`. Deşi `clone()` poate fi utilizat ca atare, este recomandată folosirea funcţiei `fork()` sau a celor din Pthreads pentru a nu crea probleme de portabilitate.

Între cele două tipuri de implementări există anumite diferenţe importante care se leagă în principal de consumul de resurse de calcul şi de siguranţă. În acest caz, prin siguranţă se au în vedere două aspecte: cât de susceptibilă e paradigma de programare la anumite tipuri de erori şi cât de bine pot fi izolate procesele pentru ca o eroare apărută în unul să nu le afecteze şi pe celelalte.

Privind din acest punct de vedere se vor enumera anumite avantaje şi dezavantaje ale celor două abordări:

Avantaje ale `fork()` faţă de Pthreads:

- `fork()` este în general o metodă mai puţin complexă din punct de vedere al efortului de programare şi dă naştere unui cod mai uşor de întreţinut;
- pentru că fiecare proces pornit prin `fork()` este separat de celelalte (are un ID şi memorie proprii), pierderile de memorie sau erorile vor fi limitate la procesul care le produce şi, teoretic, pot fi izolate;
- codurile care implementează `fork()` sunt în general mai uşor de corectat şi depanat şi sunt mai portabile;
- folosind thread-uri, există riscul apariţiei condiţiilor de cursă dacă memoria comună este citită/scrisă în paralel de mai multe fire de execuţie;
- thread-urile nu pot apela decât funcţii care sunt *thread safe* şi care au fost implementate în aşa fel încât să poată fi utilizate simultan de mai multe fire de execuţie (cu alte cuvinte, să nu folosească date stocate global);

Avantaje ale Pthreads faţă de `fork()`:

- implementările care folosesc `fork()` au în general un consum de resurse mai ridicat (din cauza memoriei şi spaţiului de adrese separate pentru fiecare proces);

- comunicarea între procesele create cu `fork()` este mai complicată, pentru că acestea nu au o zonă de memorie comună, şi consumă şi mai multe resurse de calcul decât o alternativă Pthreads;
- procesele bazate pe Pthreads sunt considerate în general mai rapide din cauza amprentei reduse asupra resurselor sistemului care duce la un timp mult mai redus de pornire şi oprire a proceselor;
- un alt avantaj al eficienţei de calcul în favoarea Pthreads este cel al comutării de context care, în cazul `fork()`, constă în comutări de proces care sunt acţiuni mult mai costisitoare din punct de vedere al timpului de calcul decât comutările de thread-uri (comutările de context se referă la salvarea unei anumite stări din execuţia procesului pentru a-l putea întrerupe şi, după aceea, pentru a-i putea continua execuţia la un moment ulterior de timp);
- deşi poate fi considerată o modalitate de eficientizare în cazul proceselor generate cu `fork()`, se foloseşte o abordare de tip *copy on write* (procesul nu copiază imediat memoria părintelui ci numai la momentul modificării acesteia) care poate duce la un efect de tip *fork bomb* dacă multe procese create vor să-şi facă propria copie de memorie în acelaşi timp;

Există însă şi aplicaţii care folosesc o abordare hibridă de combinare a acestor două modalităţi pentru a profita de avantajele amândurora. Un exemplu este serverul HTTP Apache care poate folosi mai multe procese copil independente, fiecare cu un număr de thread-uri care gestionează conexiuni separate. Această abordare profită de avantajul resurselor reduse folosite de thread-uri dar limitează efectele negative care pot apărea la nivel de proces (păstrând integritatea serverului). Desigur, această abordare mixtă implică şi un grad ridicat de complexitate care trebuie gestionată cu atenţie.

1.4 Comunicare între procese: Pipes şi Sockets

În secţiunea anterioară au fost prezentate două modalităţi de generare a proceselor paralele, fie generând procese independente, cu funcţia `fork()`, fie folosind thread-uri. Aceste două abordări nu sunt însă suficiente pentru a putea de vorbi cu adevărat de calcul paralel pentru că nu pun la dispoziţie modalităţi flexibile de comunicare. Deşi în cazul thread-urilor s-a putut obţine un tip de comunicare bazată pe memorie comună (care de altfel este cea mai performantă din punct de vedere al vitezei şi resurselor implicate), aceasta nu este posibilă decât dacă thread-urile împart

fizic memoria (adică dacă rulează pe acelaşi calculator). Problema este că sistemele de calcul paralel actuale sunt alcătuite din calculatoare cu memorie distinctă care sunt nevoite să comunice printr-o reţea. Din fericire, există numeroase opţiuni de comunicare, locale sau prin reţea, dintre care cele mai des întâlnite sunt [1]:

- *pipe*-uri care sunt canale de comunicare locală între procese sau thread-uri şi care pot fi temporare (*anonymous pipes*) sau canale permanente (*named pipes*), caz în care pot exista şi după încheierea procesului care le creează;
- *socket*-urile sunt similare cu pipe-urile numai că, pe lângă posibilitatea comunicării locale, permit şi comunicare prin reţea între maşini diferite.

1.4.1 Pipes

Pentru deschiderea unui canal de comunicare de tip pipe se foloseşte funcţia:

```
int pipe( int pipefd[2]);
```

care este definită în *unistd.h* şi care primeşte ca parametru doi descriptori de fişier (*file descriptors*) sub forma unui şir. Descriptorii de fişier au rolul celor două capete ale `pipe`-ului, adică intrarea şi ieşirea. Dacă `pipe()` execută cu succes va întoarce valoarea zero iar dacă se confruntă cu o eroare va întoarce valoarea -1. Funcţiile necesare controlării fluxului de date prin pipe sunt:

```
ssize_t write(int fd, void *buf, size_t count);

ssize_t read(int fd, void *buf, size_t count);

int close(int fd);
```

toate definite în *unistd.h*. Acestea au anumite roluri: de a trimite date prin pipe, în cazul lui `write()`, ai cărei parametrii de intrare sunt descriptorul de fişier (`fd`), adresa buffer-ului care conţine mesajul (`buf`) şi dimensiunea mesajului trimis (`count`), în octeţi; de a citi datele trimise (funcţia `read()` este de tip blocking); de a închide unul dintre descriptorii de fişier astfel încât `pipe`-ul să fie unidirecţionat (adică un proces să scrie iar altul să citească datele transmise prin el).

Următorul exemplu arată cum se poate deschide o cale de comunicare între două procese folosind un pipe anonim:

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include <sys/types.h>
#include <sys/wait.h>

//Header in care sunt definite constantele implementate.
//In acest caz, ne intereseaza dimensiunea maxima a
//bufferului unui pipe.
#include <limits.h>

int main(void)
{
  int i, status;
  pid_t my_pid;
  int size;
  char *buf;

  //Un sir stocheaza descriptorii de fisier (file descriptors)
  //care indica cele doua capete ale pipe-ului
  //(intrare si iesire).
  int pipefd[2];

  //Este creat canalul de comunicare (pipe-ul).
  if(pipe(pipefd)!=0)
  {
    perror("Pipe error\n");
    exit(1);
  }

  //Este creat noul proces.
  if( (my_pid = fork()) < 0)
  {
    perror("Fork error\n");
    exit(1);
  }
```

```
//Daca este procesul copil:
if(my_pid==0)
{
  //Procesul copil isi inchide descriptorul de scriere.
  close(pipefd[1]);

  //Asteapta sa primeasca un mesaj prin functia de tip
  //blocking read(). In acest caz asteapta doar pentru a
  //nu afisa in terminal mesaje inaintea procesului parinte.
  if(read(pipefd[0], &i, sizeof(int))<0)
  {
    perror("Read error\n");
    exit(1);
  }

  printf("PID %i receiving size\n", getpid());

  //Primeste dimensiunea mesajului.
  if(read(pipefd[0], &size, sizeof(int))<0)
  {
    perror("Read error\n");
    exit(1);
  }

  printf("From PID=%i -> Size is %i\n",getpid(),size);

  //Aloca memorie pentru stocarea mesajului. Cum strlen()
  //nu numara si caracterul de terminare al sirului, memoria
  //necesara acestuia trebuie adaugata prin 'size+1'.
  buf=malloc((size+1)*sizeof(char));

  if(read(pipefd[0], buf, size)<0)
  {
    perror("Read error\n");
    exit(1);
  }
  printf("From PID=%i -> Message is: %s\n",getpid(), buf);

  //Elibereaza memoria.
  free(buf);
  close(pipefd[0]);
```

```
}

//Daca este procesul parinte:
if(my_pid!=0)
{
  //Procesul parinte isi inchide descriptorul de citire.
  close(pipefd[0]);

  //Afiseaza dimensiunea maxima a mesajului care poate
  //fi scris. Aceasta limita este data de dimensiunea
  //buffer-ului unui pipe. Dimensiunea este buffer-2
  //pentru a avea loc pentru caracterul de linie noua
  //care apare cand este apasat 'enter' si pentru
  //caracterul de terminare a sirului.
  printf("Write something shorter than %li characters"
         "and press enter\n", PIPE_BUF/sizeof(char)-2);

  //Aloca memoria pentru buffer si citeste input-ul.
  buf=malloc((int)PIPE_BUF);
  fgets(buf,(int)PIPE_BUF-2, stdin);
  size=strlen(buf);

  //Da semnalul de start procesului copil.
  if( write(pipefd[1], &i, sizeof(int)) <0)
  {
    perror("Write error\n");
    exit(1);
  }

  //Trimite dimensiunea sirului.
  if( write(pipefd[1], &size, sizeof(int)) <0)
  {
    perror("Write error\n");
    exit(1);
  }

  //Trimite sirul.
  if( write(pipefd[1], buf, size) <0)
  {
    perror("Write error\n");
    exit(1);
  }
```

```
    free(buf);
    close(pipefd[1]);
  }

  //Procesul parinte va fi singurul cu variabila my_pid
  //nenula si va astepta incheierea procesului copil.
  if(my_pid!=0)
  {
       if (wait(&status) < 0)
       perror("Wait error");
       _exit(1);
  }

  return 0;
}
```

1.4.2 Sockets

Venind ca o extindere a conceptul de *pipe*, *socket*-ul este o modalitate mai flexibilă de comunicare ce poate fi folosită şi la trimiterea datelor între maşini de calcul distincte conectate la o reţea locală sau la internet (folosind protocolul IP). La nivelul de transport (*transport layer*) *socket*-urile au un nivel ridicat de flexibilitate la nivelul protocolului folosit: de tip flux de date (*stream sockets*) - bazate pe protocolul TCP (*Transmission Control Protocol*), caz în care livrarea pachetelor este garantată şi este asigurată automat de protocolul TCP; *datagram sockets* - foloseşte protocolul UDP (*User Datagram Protocol*), caz în care livrarea pachetelor nu este garantată; *brute* - permit trimiterea şi primirea pachetelor fără un protocol la nivelul de transport; cu transmitere secvenţială (*sequenced packet sockets*) - în plus faţă de *socket*-urile de tip flux, păstrează limitele mesajelor (în cazul TCP mai multe mesajele pot fi concatenate sub unul singur, pierzându-se graniţa lor). Comunicarea are de obicei loc între un server şi clienţi. Serverul poate satisface cererile venite de la clienţi secvenţial sau paralel (de exemplu generând un nou proces prin `fork()` sau un nou thread care să comunice cu fiecare client). Etapele generale care sunt necesare pentru construirea unui client sunt: crearea unui socket prin apelul de sistem `socket()`, conectarea socket-ului la adresa serverului cu apelul de sistem `connect()` şi trimiterea/recepţionarea datelor prin `write()/read()` sau `send()/recv()`. Un server are nevoie de: crearea unui *socket*

cu funcţia `socket()`, legarea *socket*-ului la adresa serverului, aşteptarea conexiunilor de la clienţi cu `listen()`, acceptarea conexiunilor folosind apelul de sistem `accept()` şi trimiterea/recepţionarea datelor.

Funcţiile necesare dezvoltării unui client/server sunt:

- `int socket( int af, int type, int protocol);`

(creează un *socket*; `af` - familia de adrese folosită; `type` - tipul de comunicare (stream, datagram, sequenced packet); protocol - valoarea zero indică folosirea protocolului implicit pentru tipul de comunicare selectat, iar o valoare nenulă specifică în mod explicit protocolul)

- `int bind( int sockfd, const struct sockaddr *addr,`
  `socklen_t addrlen);`

(`sockfd` - descriptorul de fişier întors de `socket()`; `addr` - adresa serverului; `addrlen` - dimensiunea adresei)

- `int listen( int sockfd, int backlog);`

(`backlog` - numărul maxim de conexiuni în aşteptare care să fie acceptate)

- `int accept( int sockfd, struct sockaddr *addr,`
  `socklen_t *addrlen);`

- `ssize_t send( int sockfd, const void *buf, size_t len,`
  `int flags);`

(`buf` - mesajul care va fi trimis; `len` - lungimea mesajului; `flags` - diverse opţiuni de transmitere)

- `ssize_t recv( int sockfd, void *buf, size_t len, int flags);`

- `int inet_pton(int af, const char *src, void *dst);`

(converteşte şirul de caractere `src` sub forma structurii unei adrese de reţea indicată de familia `af`; definită în *arpa/inet.h*; `src` - adresa serverului sub forma unui şir de caractere; `dst` - destinaţia unde este copiată adresa sub forma indicată de `af`)

- `int connect( int sockfd, const struct sockaddr *addr,`

```
        socklen_t addrlen);
```

În exemplul următor este creat un server care distribuie o acţiune (în acest caz adunarea elementelor unei matrici) unor alte procese de tip client. Mai exact, serverul aşteaptă conectarea unui număr de clienţi predeterminat şi le trimite acestora dimensiunea matricii pe care urmează să o primească şi după aceea trimite matricea în sine. După distribuirea informaţiei, clienţii adună elementele matricii primite şi trimit înapoi serverului rezultatul. Serverul adună valorile primite şi afişează valoarea totală. Pentru exemplul de faţă a fost folosită adresa locală (*localhost*), dar în cazul folosirii mai multor maşini conectate prin reţea, aceasta trebuie schimbată cu IP-ul serverului. Serverul este procesul care va fi pornit primul, urmând să fie porniţi cu un număr de `NO_CLIENTS` clienţi.

Serverul:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>

#define PORT 8000

//Numarul de clienti pe care ii asteapta sa se conecteze.
#define NO_CLIENTS 5

int main(void)
{
 int i,j;
 double sum=0;
 double temp_sum;
 int my_socket;

 //Va contine socket-urile prin care se conecteaza clientii.
 int accept_fd[NO_CLIENTS];

 //Adresa serverului.
```

```
struct sockaddr_in server_address;
//Adresa clientului.
struct sockaddr_in client_address;

//Dimensiunea adresei clientului.
socklen_t cli_add_len;

int n_matrices=NO_CLIENTS; //Numarul de matrici.
int matrix_size=10*10;//Dimensiunea unei matrici.
double **matrices;//Fiecare matrice este definita ca un sir.

//Se aloca memoria pentru stocarea matricilor.
if( (matrices=malloc(n_matrices*sizeof(double *))) ==NULL)
{
  perror("Malloc error\n");
  exit(1);
}

//Matricile sunt initializate cu valoarea 1.
for(i=0;i<n_matrices;i++)
{
  matrices[i]=malloc(matrix_size*matrix_size
                     *(sizeof(double)));

  for(j=0;j<matrix_size*matrix_size;j++)
  {
    matrices[i][j]=1;
  }
}

//Se creeaza socket-ul.
if((my_socket=socket(AF_INET,SOCK_STREAM,0)) <0)
{
  perror("Socket error\n");
  exit(1);
}

//Memoria adresei este initializata cu 0.
memset(&server_address,0,sizeof(server_address));
//Familia adresei este IPV4
server_address.sin_family=AF_INET;
```

```
//Htonl() converteste un int primit ca input in
//format binar pentru retea.
//Parametrul INADDR_ANI ii indica socket-ului sa asculte
//toate interfetele disponibile.
server_address.sin_addr.s_addr=htonl(INADDR_ANY);

//Htons() converteste un short int primit ca input in
//format binar pentru retea.
server_address.sin_port=htons(PORT);

//Se leaga adresa de port.
if(bind(my_socket,(struct sockaddr *)&server_address,
   sizeof(server_address)) <0)
{
  perror("Bind error\n");
  exit(1);
}

//Socket-ul este indicat ca disponibil pentru a accepta
//conexiuni. Va accepta maxim 5 conexiuni intre
//accept-uri. Daca numarul de conexiuni e depasit, acestea
//vor fi refuzate.
if(listen(my_socket,5) <0)
{
  perror("Listen error\n");
  exit(1);
}

printf("Waiting for clients\n");

//Serverul intai accepta conexiunile de la clienti.
for(i=0; i<n_matrices;i++)
{
  //Se determina marimea adresei clientului.
  cli_add_len=(socklen_t)sizeof(client_address);

  //Conexiunile sunt acceptate.
  accept_fd[i]=accept(my_socket,
                      (struct sockaddr*)&client_address,
                       &cli_add_len);

  if (accept_fd[i] < 0)
```

```
  {
    perror("Accept error\n");
    exit(1);
  }

  printf("Connected to client %i out of %i\n",
         i+1,NO_CLIENTS);
}

printf("Sending data\n");

//Dupa ce s-au facut conexiunile cu clientii, serverul
//le trimite fiecaruia dimensiunea matricii si
//continutul ei.
for(i=0;i<n_matrices;i++)
{
  if(send(accept_fd[i],&matrix_size,sizeof(int),0) <0)
  {
    perror("Send error\n");
    exit(1);
  }

  if(send(accept_fd[i],(void *)matrices[i],
          matrix_size*sizeof(double),0) <0)
  {
    perror("Send error\n");
    exit(1);
  }
}

printf("Receiving data\n");

//Serverul primeste valorile calculate de clienti.
for(i=0;i<n_matrices;i++)
{
  if(recv(accept_fd[i], &temp_sum, sizeof(double),0) <0)
  {
    perror("Recv error\n");
    exit(1);
  }

  sum+=temp_sum;
```

```
 }

 printf("Total sum is %lf\n",sum);

 //Se elibereaza memoria.
 for(i=0;i<n_matrices;i++)
 {
   free(matrices[i]);
 }
 free(matrices);

 return 0;
}
```

Clientul:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<string.h>
#include<stdlib.h>
#include<netinet/in.h>
#include<arpa/inet.h>

#define PORT 8000

//Adresa locala - localhost.
#define SERVER_IP "127.0.0.1"

int main(void)
{
 int i;
 double sum=0;
 int matrix_size;
 double *matrix;
 int my_socket;

 //Adresa serverului.
 struct sockaddr_in server_address;

 //Se creeaza socket-ul.
 if((my_socket=socket(AF_INET,SOCK_STREAM,0)) <0)
```

```
{
  perror("Socket error\n");
  exit(1);
}

memset(&server_address,0,sizeof(server_address));
server_address.sin_family=AF_INET;
server_address.sin_port=htons(PORT);

//Converteste adrese IPv4 si IPv6 din format text in
//format binar pentru retea.
if(inet_pton(AF_INET, SERVER_IP,
             &server_address.sin_addr)<=0)
{
  perror("Pton error\n");
  exit(1);
}

//Conecteaza socket-ul (my_socket) la adresa
//serverului a carei dimensiune este sockaddr_in.
if(connect(my_socket,(struct sockaddr *)&server_address,
           sizeof(struct sockaddr_in)) <0)
{
  perror("Connect error\n");
  exit(1);
}

 printf("Connected\n");

 //Clientul primeste dimensiunea matricii.
 if(recv(my_socket,&matrix_size,sizeof(int),0) <0)
  {
    perror("Recv error\n");
    exit(1);
  }

 printf("Matrix size %i\n",matrix_size);

 if((matrix=malloc(matrix_size*sizeof(double)))==NULL)
 {
   perror("Malloc error\n");
   exit(1);
```

```
  }

  //Clientul primeste matricea.
  if(recv(my_socket,matrix,matrix_size*sizeof(double),0) <0)
  {
     perror("Recv error\n");
     exit(1);
  }

  //Clientul aduna toate elementele matricii.
  for(i=0;i<matrix_size;i++) sum+=matrix[i];

  //Clientul trimite serverului suma elementelor.
  if(send(my_socket, &sum, sizeof(double),0) <0)
  {
     perror("Send error\n");
     exit(1);
  }

 free(matrix);

 return 0;
}
```

Serverul va afişa:

Waiting for clients
Connected to client 1 out of 5
Connected to client 2 out of 5
Connected to client 3 out of 5
Connected to client 4 out of 5
Connected to client 5 out of 5
Sending data
Receiving data
Total sum is 500.000000

în timp ce fiecare client va afişa:

Connected
Matrix size 100

Capitolul 2

Interfaţa MPI – Elemente introductive

2.1 Concepte de bază în calcul paralel

Noţiunea de calcul paralel presupune utilizarea simultană a resurselor de calcul multiple pentru rezolvarea unei probleme de calcul. Într-un cod de calcul paralel problema este partajată în sub-probleme, denumite în continuare sarcini. În cadrul unui sistem multi-procesor, sarcinile se pot distribui pe diferite noduri de calcul, fiind efectuate în paralel. Nodurile de calcul reprezintă unităţi fizice de calcul ce corespund de obicei miezurilor din sistemul multi-procesor.

2.1.1 Avantajele calculului paralel

Raportul între timpul de calcul paralel (t_p) şi timpul de calcul serial (t_s) se numeşte *speedup*, $\mathcal{S} = t_s/t_p$. În condiţii ideale, având la dispoziţie N noduri de calcul, timpul de calcul se reduce de N ori. În realitate $\mathcal{S} < N$, datorită timpului suplimentar de comunicare între noduri, cât şi datorită faptului că în anumite situaţii dependinţele între diferitele sarcini pot cauza întârzieri.

2.1.2 Iniţializarea şi finalizarea mediului MPI

Pentru desfăşurarea calculelor de tip paralel se pot folosi diferite interfeţe (OpenMPI - *Message Passing Interface*, MPICH) către limbajele de programare C/C++ şi Fortran90.

În continuare vom discuta interfaţa MPI [2–5], utilizând limbajul de programare C. Pornind de la structura de bază a unui cod C, iniţializarea mediului

MPI se face prin apelarea funcţiei

```
MPI_Init( int *argc, char ***argv);
```

adică

```
int main( int argc, char **argv)
{
   MPI_Init( &argc, &argv);
}
```

în care *argc* reprezintă un pointer către numărul de argumente, iar *argv* este un pointer către vectorul de argumente.

Pentru finalizarea mediului MPI se apelează:

```
MPI_Finalize()
```

2.1.3 Rangul unui proces

Sarcinile din cadrul calculului paralel sunt distribuite pe un număr de *procese*. Fiecăruia i se ataşează un identificator, denumit *rangul procesului*. Acestea sunt numere întregi începând cu zero. Numărul de procese este specificat de obicei la momentul execuţiei programului paralel, dar poate fi modificat şi pe parcurs. Procesele pot schimba informaţii prin operaţii de comunicare. Atribuirea rangurilor face posibil ca unele procese să efectueze anumite secvenţe de cod.

2.1.4 Comunicatorul `MPI_COMM_WORLD`

Un comunicator reprezintă un grup de procese. Rolul unui comunicator este acela de a limita comunicarea la un anumit sub-set de procese. Comunicatorul standard, care conţine toate procesele, se numeşte `MPI_COMM_WORLD`. Este de subliniat faptul că rangul unui proces este specificat în cadrul unui comunicator.

Orice proces îşi poate extrage valoarea propriului rang folosind funcţia:

```
int MPI_Comm_rank( MPI_Comm comm, int *rank);
```

Dimensiunea comunicatorului se poate extrage folosind funcţia:

```
int MPI_Comm_size( MPI_Comm comm, int *size);
```

Prin apelul celor două funcţii este returnat rangul procesului *rank* şi dimensiunea comunicatorului *size*:

```
MPI_Comm_rank( MPI_COMM_WORLD, &rank);
MPI_Comm_size( MPI_COMM_WORLD, &size);
```

2.1.5 Compilare şi rulare

Pentru compilare se foloseşte compilatorul C aferent interfeţei OpenMPI:

mpicc.openmpi -o program program.c

având drept rezultat creearea fişierului executabil *program* din codul sursă *program.c*.

Rularea programului paralel se face folosind comanda:

mpiexec -n 8 ./program

în care am specificat, spre exemplu, execuţia paralelă pe 8 noduri.

2.2 Program MPI minimal – Hello world!

Vom considera un program MPI minimal, în care se pun în evidenţă elementele discutate anterior, şi anume: iniţializarea mediului MPI, procesele din comunicatorul standard `MPI_COMM_WORLD` şi finalizarea mediului MPI.

Se introduce fişierul de tip header specific *mpi.h*, pe lângă *stdlib.h* şi *stdio.h* corespunzătoare unui cod C serial. Variabilele *numprocs* şi *myrank* sunt asociate numărului de procese din comunicator şi rangului procesului.

```
#include<stdlib.h>
#include<mpi.h>
#include<stdio.h>

int main( int argc, char **argv)
{
  // numarul de procese din comunicator
  int numprocs;
  // rangul procesului
  int myrank;

  // Initializare MPI
  MPI_Init( &argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank);

  // Program
  printf( "Eu sunt nodul %i \n", myrank);

  // Finalizare MPI
  MPI_Finalize();

  return 0;
}
```

În urma rulării programului se obţine un rezultat de tipul:

Eu sunt nodul 0
Eu sunt nodul 2
Eu sunt nodul 3
Eu sunt nodul 1

Fiecare nod afişează pe ecran un string urmat de rangul procesului corespunzător. Se observă că ordinea în care sunt afişate rangurile nu este unică şi se poate schimba rulând succesiv programul.

Capitolul 3

Operaţii de comunicare point-to-point

Operaţiile de comunicare permit schimbul de date între procese. În continuare vom discuta operaţiile de comunicare *point-to-point* în care sunt implicate perechi de procese [2, 3].

3.1 Funcţiile de comunicare `MPI_Send()` şi `MPI_Recv()`

Un prim exemplu de operaţii de comunicare *point-to-point* este reprezentat de funcţiile `MPI_Send()` şi `MPI_Recv()`.

Pentru a transmite un mesaj, rangul i (nodul sursă) va apela funcţia `MPI_Send()`, în timp ce rangul j (nodul destinaţie) va prelua datele apelând funcţia `MPI_Recv()`. Detaliem în continuare prototipurile celor doua funcţii.

```
int MPI_Send( void *smessage, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm);
```

- `smessage`: specifică un buffer care conţine elementele de date care urmează să fie transmise;
- `count`: numărul de elemente care urmează să fie transmise;
- `datatype`: specifică tipul de date din buffer; toate datele au acelaşi tip;
- `dest`: specifică rangul procesului care urmează să primească datele;
- `tag`: o etichetă care permite destinatarului să distingă între diferitele mesaje de la aceeaşi sursă;
- `comm`: specifică comunicatorul folosit în comunicare;

```
int MPI_Recv( void *rmessage, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- rmessage: specifică bufferul în care se primesc datele;
- count: numărul maxim de elemente care urmează să fie primite;
- datatype: tipul datelor care vor fi primite;
- source: specifică rangul procesului care trimite mesajul;
- tag: eticheta pe care trebuie s-o aibe mesajul pentru a fi primit;
- comm: specifică comunicator-ul folosit în comunicare;
- status: este o structură care specifică informaţii despre mesaj, după încheierea operaţiei de comunicare (id_procesor send, cod de eroare);

Observaţii:

- Instrucţiunile MPI_Send şi MPI_Recv sunt de tip *blocking*.
- Mărimea mesajului poate fi calculată înmulţind numărul count cu numărul de octeţi corespunzător tipului datatype.
- Tag-ul este un număr întreg.
- Valoarea predefinită pentru variabila source, MPI_ANY_SOURCE, se poate utiliza în cadrul funcţiei MPI_Recv în cazul în care nodul sursă nu este cunoscut.
- Valoarea predefinită pentru variabila tag, MPI_ANY_TAG, specifică faptul că un proces poate primi un mesaj, indiferent de etichetă.

Mai jos este indicat un exemplu de cod:

```
#include<stdlib.h>
#include<mpi.h>
#include<stdio.h>

int main( int argc, char **argv)
{
  // numarul de procese din comunicator
  int numprocs;
  // rangul procesului
  int rank;
```

```
    // Initializare MPI
    MPI_Init( &argc, &argv);
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);

    // Program
    // Nodul 0 trimite un numar intreg x catre nodul 1
    if (rank==0) {
        int x = 2;
        MPI_Send( &x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    // Nodul 1 primeste numarul intreg x de la nodul 0
    if (rank==1) {
        int x;
        MPI_Recv( &x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        printf( "Nodul %i: x = %i \n", rank, x);
    }

    // Finalizare MPI
    MPI_Finalize();

    return 0;
}
```

3.2 Ordinea operaţiilor in MPI

În MPI mesajele sunt trimise în ordinea care este specificată în cod. Dacă avem 2 operaţii de tip `MPI_Send` succesive într-un proces şi 2 operaţii `MPI_Recv` în alt proces, ordinea transmiterii mesajelor se va păstra. În schimb, dacă avem mai multe procese implicate, se poate întâmpla ca ordinea să nu se păstreze. În secvenţa de cod următoare două seturi de date, *send_ data1* şi *send_ data2*, sunt trimise de la nodul 0 către 2 pe căi diferite. În timp ce *send_ data1* este trimis direct către nodul 2, *send_ data2* este mai întâi trimis către nodul 1, unde este recepţionat şi trimis mai departe către nodul 2. Nodul 2 va primi cele două seturi de date, însă întrucât nodul de la care primeşte este specificat prin `MPI_ANY_SOURCE` este posibil ca primirea datelor să se efectueze în ordinea trimiterii lor la nodul 0, dar şi invers [2].

```
MPI_Comm_rank( MPI_COMM_WORLD, &rank);
if (rank==0) {
    MPI_Send( send_data1, n, MPI_INT, 2, 0, MPI_COMM_WORLD);
    MPI_Send( send_data2, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
else if (rank==1) {
    MPI_Recv( recv_data1, n, MPI_INT, 0, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
    MPI_Send( recv_data1, n, MPI_INT, 2, 0, MPI_COMM_WORLD);
}
else if (rank==2) {
    MPI_Recv( recv_data1, n, MPI_INT, MPI_ANY_SOURCE, 0,
       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv( recv_data2, n, MPI_INT, MPI_ANY_SOURCE, 0,
       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Ordinea operaţiilor de tip *Send/Receive* este importantă întrucât este posibilă apariţia unor situaţii de *deadlock*. În exemplul următor fiecare proces execută mai întâi o operaţie de tip *Receive*, astfel încât fiecare va aştepta date indefinit.

```
if (rank==0) {
    MPI_Recv( recv_data2, n, MPI_INT, 1, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
    MPI_Send( send_data1, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
if (rank==1) {
    MPI_Recv( recv_data1, n, MPI_INT, 0, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
    MPI_Send( send_data2, n, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

În funcţie de implementare, se pot folosi buffere de sistem, iar datele stocate pot fi preluate ulterior. În exemplul următor, operaţiile *Send* sunt de tip blocking. Cu toate acestea, situaţia de deadlock este evitată dacă sunt utilizate bufferele sistemului.

```
if (rank==0) {
    MPI_Send( send_data1, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( recv_data2, n, MPI_INT, 1, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
```

```
}
if (rank==1) {
    MPI_Send( send_data2, n, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Recv( recv_data1, n, MPI_INT, 0, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
}
```

Un fragment de cod care nu produce situaţii de deadlock, chiar dacă nu sunt folosite bufferele sistemului este următorul:

```
if (rank==0) {
    MPI_Send( send_data1, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( recv_data2, n, MPI_INT, 1, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
}
if (rank==1) {
    MPI_Recv( recv_data1, n, MPI_INT, 0, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
    MPI_Send( send_data2, n, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

În general, o implementare sigură se poate obţine dacă rangurile pare execută *Send/Receive*, iar cele impare *Receive/Send*.

3.3 Rutine de comunicare de tip non-blocking

Folosirea operaţiilor de tip blocking poate conduce la timpi de aşteptare mari. Spre exemplu, un proces care execută o operaţie *Send* de tip blocking va aştepta până când bufferul *send* este copiat în bufferul sistemului sau în cel de tip *receive*. Se aşteaptă apoi ca mesajul să ajungă la destinatar. Este preferabil adesea ca timpul de aşteptare să fie folosit la efectuarea unor calcule, de unde rezultă utilitatea folosirii unor operaţii de comunicare de tip non-blocking. Astfel se pot suprapune operaţiile de calcul cu cele de comunicare [2].

O instrucţiune de tip non-blocking iniţiază o operaţie de tip *Send* şi returnează imediat controlul procesului care trimite date. Bufferul *send* nu poate fi reutilizat în condiţii de siguranţă întrucât transmisia mesajului poate fi încă în progres. Este necesară o altă operaţie pentru a confirma ulterior completarea transmisiei.

Prototipul operaţiei non-blocking de tip *Send* este:

```
int MPI_Isend( void *buffer, int count, MPI_Datatype type, int dest,
               int tag, MPI_Comm comm, MPI_Request *request)
```

Observaţii:

- funcţia returnează imediat controlul procesului;
- este demarat procesul de trimitere a datelor;
- există o variabilă suplimentară, `request`, care poate specifica informaţii despre status-ul operaţiei de comunicare;

Similar, pentru funcţia de *Receive* non-blocking avem prototipul:

```
int MPI_Irecv( void *buffer, int count, MPI_Datatype type, int source,
               int tag, MPI_Comm comm, MPI_Request *request)
```

Înaintea refolosirii bufferului de o altă operaţie de comunicare, se testează finalizarea primei operaţii de comunicare prin intermediul variabilei `request`.

```
int MPI_Test( MPI_Request *request, int *flag, MPI Status *status)
```

Observaţii:

- dacă `flag`=1 (true) atunci operaţia este finalizată;
- `status`: similar operaţiei `MPI_Recv()`; este nedefinit dacă operaţia nu este finalizată;
- dacă `request` se referă la o operaţie de tip Send, `status.MPI_ERROR` este nedefinită.

Procesul care apelează funcţia

```
int MPI_Wait( MPI_Request *request, MPI_Status *status)
```

este blocat până când operaţia identificată prin `request` este finalizată. Deci, un buffer poate fi reutilizat după ce operaţia `MPI_Wait` returnează. Se pot folosi în

pereche `MPI_Send`/`MPI_IRecv` şi invers.

Mai jos este prezentat un exemplu de cod care utilizează operaţii de comunicare de tip non-blocking.

```
#include<stdlib.h>
#include<mpi.h>
#include<stdio.h>

int main( int argc, char **argv)
{
  // numarul de procese din comunicator
  int numprocs;
  // rangul procesului
  int rank;
  MPI_Status status;
  MPI_Request request;
  int data;

  // Initializare MPI
  MPI_Init( &argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank);

  request = MPI_REQUEST_NULL;

  // Program
  if(rank==0){
      data = 13;
      MPI_Isend( &data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &
          request);
      // Operatii de calcul
      printf( "inainte de MPI_Wait: Nodul %i a trimis %i\n",
          rank, data);
  }
  if(rank==1){
      MPI_Irecv( &data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &
          request);
      // Operatii de calcul
      printf( "inainte de MPI_Wait: Nodul %i a primit %i\n",
          rank, data);
```

```
    }

    MPI_Wait( &request, &status);

    if(rank==0){
        printf("dupa MPI_Wait: Nodul %i a trimis %i\n", rank,
            data);
    }
    if(rank==1){
        printf("dupa MPI_Wait: Nodul %i a primit %i\n", rank,
            data);
    }

    // Finalizare MPI
    MPI_Finalize();

    return 0;
}
```

În urma rulării codului, se obţine:

înainte de MPI_Wait: Nodul 0 a trimis 13
după MPI_Wait: Nodul 0 a trimis 13
înainte de MPI_Wait: Nodul 1 a primit 0
după MPI_Wait: Nodul 1 a primit 13

Se observă că, în urma apelurilor funcţiilor `MPI_ISend` şi `MPI_IRecv`, programul continuă cu funcţiile următoare. Sunt afişate pe ecran valorile pentru variabila *data* înainte şi după apelul funcţiei `MPI_Wait()`. Astfel se pune în evidenţă posibilitatea utilizării timpului de calcul pentru nodul 0, până când nodul 1 primeşte valoarea trimisă. După apelul funcţiei `MPI_Wait()` comunicarea este efectuată.

3.4 Tipuri de date

În MPI sunt predefinite tipuri de date, în mod similar limbajelor de programare C/C++, Fortran etc. O listă a tipurilor de date MPI este indicată în tabelul 3.1, prin comparaţie cu cele din limbajul de programare C [2, 3].

Tip de date MPI	Tip de date C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	fără corespondent în C
MPI_BYTE	byte

Tabela 3.1: Tipuri de date MPI.

MPI oferă posibilitatea ca, folosind tipurile de date pre-existente, să se construiască tipuri de date derivate cu ajutorul unor constructori. Crearea unui tip de date derivat se face la momentul rulării aplicaţiei în două etape:

- definirea propriu-zisă a tipului nou de date prin apelul unei funcţii specifice;
- crearea ("comiterea") noului tip de date prin apelul funcţiei:

```
int MPI_Type_commit( MPI_Datatype * newdatatype);
```

Câteva exemple de funcţii pentru definirea tipurilor de date derivate :

- `int MPI_Type_contiguous( int count, MPI_Datatype old_type, MPI_Datatype *newtype)` - tipul **contiguu** - produce un nou tip de date făcând mai multe copii ale unui tip existent, cu deplasări care sunt multipli ai extensiei tipului vechi.
- `int MPI_Type_vector( int count, int blocklength, int stride, MPI_Datatype old_type, MPI_Datatype *newtype )` - tipul **vector** - permite specificarea unor date situate în zone necontigue de memorie. Elementele tipului vechi pot fi separate între ele de spaţii având lungimea egală cu un multiplu al extinderii tipului (deci cu un pas constant).

- `int MPI_Type_hvector( int count, int blocklength, MPI_Aint stride, MPI_Datatype old_type, MPI_Datatype *newtype )` - tipul **hvector** - similar tipului vector, cu diferenţa că pasul se dă în număr de octeţi şi nu în număr de elemente.

- `int MPI_Type_indexed( int count, int array_of_blocklengths[], int array_of_indices[], MPI_Datatype old_type, MPI_Datatype *newtype )` - tipul **indexed** - în acest caz fiecare bloc are un număr diferit de copii ale tipului vechi şi o deplasare diferită de ale celorlalte. Pentru acest tip de date, deplasările sunt multipli ai extinderii vechiului tip.

- `int MPI_Type_hindexed( int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype old_type, MPI_Datatype *newtype )` - tipul **hindexed** - este asemănător tipului indexat, cu diferenţa că în acest caz deplasările sunt măsurate în octeţi.

- `int MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[], MPI_Datatype *newtype)` - tipul **struct** - este o generalizare a tipul hindexed prin faptul că permite ca fiecare bloc să fie alcătuit din replici ale unor tipuri de date diferite.

3.5 Aplicaţii

3.5.1 Calculul unei integrale

Considerăm o funcţie $f(x)$ definită pe intervalul $[a, b]$. Ne propunem să calculăm integrala definită

$$I = \int_a^b f(x)\, dx. \tag{3.1}$$

3.5.1.1 Metoda dreptunghiului.

Considerăm un grid arbitrar pe intervalul $[a, b]$, definit de setul $\{x_i\}$, $i = \overline{0, N}$, astfel încât $x_0 = a$ si $x_N = b$, unde N este numărul de subintervale.

Integrala definită se poate evalua prin metoda dreptunghiului, calculând

suma Riemann:

$$I = \sum_{i=0}^{N-1} f(\bar{x}_i) \cdot \Delta_i, \quad \text{unde} \quad \bar{x}_i = (x_{i+1} + x_i)/2, \quad \Delta_i = x_{i+1} - x_i \tag{3.2}$$

Dacă gridul ales este uniform, avem:

$$x_i = a + (b - a)/N \cdot i \tag{3.3}$$

În acest caz, $\Delta_i = (b-a)/N \equiv \Delta$ şi integrala devine suma valorilor funcţiei în punctele $\bar{x}_i$ multiplicată cu Δ:

$$I = \Delta \sum_{i=0}^{N-1} f(\bar{x}_i). \tag{3.4}$$

Acurateţea de calcul a integralei depinde de fineţea gridului utilizat. În cazul funcţiilor cu variaţie rapidă, este necesară utilizarea unui număr relativ mare de puncte în grid. Problema se poate paraleliza, având în vedere că intervalul $[a, b]$ se poate împărţi în N_s subintervale, $[a, c_1]$, $[c_1, c_2]$, ..., $[c_{N_s-1}, b]$, ($a = c_0$, $b = c_{N_s}$):

$$\int_a^b f(x)dx = \int_a^{c_1} f(x)dx + \int_{c_1}^{c_2} f(x)dx + \ldots + \int_{c_{N_s-1}}^{b} f(x)dx \tag{3.5}$$

Pentru fiecare din cele N_s subintervale, se aplică metoda dreptunghiului, descrisă schematic în Fig. 3.1. Din perspectiva calculului paralel, fiecare integrală pe domeniu determinat $[c_i, c_{i+1}]$ constituie câte o sarcină, S_i. Având la dispoziţie un sistem multi-procesor cu N_p noduri, putem fixa $N_s = N_p$, în asa fel încât fiecare nod de calcul va efectua o integrală cu un număr de N/N_p subintervale de dimensiune dx.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

// functia de integrat f(x)
double f( double x)
{
  return 4.*sqrt(1.-x*x);
}
```

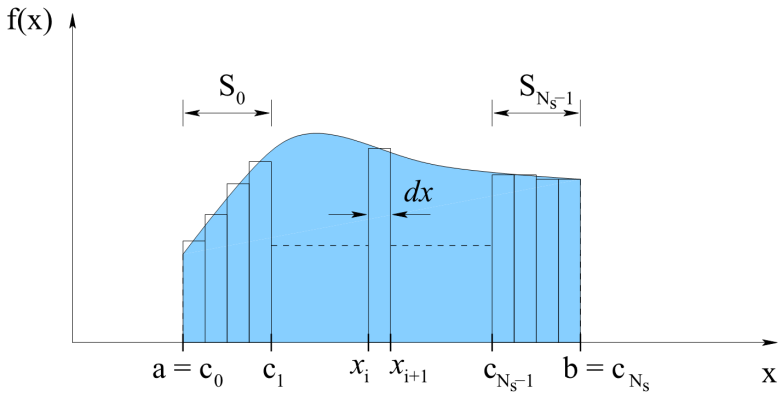

Figura 3.1: Integrare prin metoda dreptunghiului. Sarcinile reprezintă integrarea pe un subdomeniu.

```
// functia care calculeaza integrala
double calculeaza_integrala( double a, double b, long N)
{
  int i;
  double dx = (b-a)/(double)N;
  double x;
  double s;
  s = 0;
  for (i=0;i<N;i++)
      {
      x = a + (double)i * dx + dx/2.;
      s = s + f(x);
      }
  s = s * dx;
  return s;
}

int main( int argc, char **argv)
{
int Np, rank;
long i;
// limitele de integrare
double a, b;
double N;
double c1, c2;
```

```
long Ni;

MPI_Init( &argc, &argv);
MPI_Comm_size( MPI_COMM_WORLD, &Np);
MPI_Comm_rank( MPI_COMM_WORLD, &rank);

// MASTER -- rang 0
if (rank==0)
{
    double s, I;
    // input
    a = 0.;
    b = 1.;
    N = 10000;
    // trimite a, b catre procesele de tip worker
    for (i=1;i<Np;i++)
    {
         // trimite intervalul de integrare [c1,c2]
         c1 = a + (b-a)/(double)(Np-1) * (double)(i-1);
         c2 = a + (b-a)/(double)(Np-1) * (double)i;
         Ni = (long) ( (double)N/(double)(Np-1) );
         MPI_Send( &c1, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
         MPI_Send( &c2, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
         MPI_Send( &Ni, 1, MPI_LONG, i, 0, MPI_COMM_WORLD);
    }
    // primeste rezultatele integralelor si calculeaza
       rezultatul final
    for (i=1;i<Np;i++)
    {
         MPI_Recv( &s, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
            MPI_COMM_WORLD,
                   MPI_STATUS_IGNORE);
         I = I + s;
    }
    // afiseaza rezultatul integralei
    printf( "I = %f \n", I);
}

// WORKERS -- rangurile 1, 2, ... ,
else
{
double s;
```

```
// primeste intervalul de integrare
MPI_Recv( &c1, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
MPI_Recv( &c2, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
MPI_Recv( &Ni, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
// calculeaza integrala pe intervalul [c1,c2]
s = calculeaza_integrala( c1, c2, Ni);
// trimite rezultatul catre MASTER
MPI_Send( &s, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}

MPI_Finalize();

return 0;
}
```

3.5.1.2 Metoda Monte-Carlo

Integrarea prin metoda Monte-Carlo este o tehnică de integrare numerică ce foloseşte numere aleatoare. Ca şi în cazul anterior, considerăm integrala funcţiei $f(x)$ pe intervalul $[a, b]$. Vom presupune pentru simplitate că funcţia $f(x)$ este pozitiv definită pe intervalul de integrare, adică $f(x) \geq 0$ pentru orice $x \in [a, b]$.

Primul pas presupune stabilirea unui număr c astfel încât $c \geq f(x)$ pentru orice $x \in [a, b]$. În acest fel delimităm un domeniu rectangular, $[a, b] \times [0, c]$, care conţine funcţia $f(x)$ pe intervalul considerat. În continuare se extrage un număr mare (N) de perechi de numere aleatoare (x, y), cu $x \in [a, b]$ şi $y \in [0, c]$. Pentru fiecare pereche se testează condiţia $f(x) < y$, adică dacă punctul generat de extragerea numerelor aleatoare este sub grafic sau nu. Dacă această condiţie este îndeplinită, se incrementează variabila N_1. Invers, dacă punctul (x, y) se găseşte deasupra graficului, se incrementează variabila N_2. Evident, după generarea celor N perechi vom avea $N = N_1 + N_2$.

Întrucât valoarea integralei este egală cu aria conţinută între grafic şi abscisă, este proporţională cu valoarea N_1. În domeniul rectangular de arie $c \cdot (b - a)$ sunt distribuite uniform N puncte. Obţinem aşadar valoarea integralei:

$$I = \int_a^b f(x)\, dx = \frac{N_1}{N} \cdot c \cdot (b - a). \tag{3.6}$$

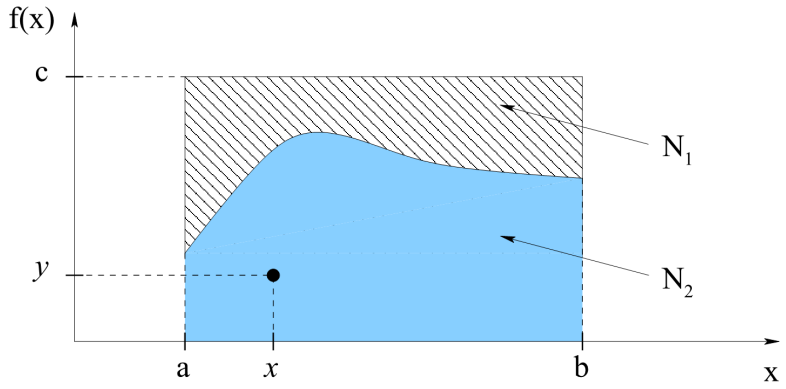

Figura 3.2: Integrare prin metoda Monte-Carlo.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
#include <mpi.h>

// functia de integrat f(x)
double f( double x)
{
  return 4.*sqrt(1.-x*x);
}

// functia care calculeaza integrala
double calculeaza_integrala( double a, double b, double c,
    long N)
{
  long i;
  const gsl_rng_type * T;
  gsl_rng * r;
  double u;
  double x, y;
  long N1=0, N2=0;
  double I;

  gsl_rng_env_setup();
```

```
  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  for (i=0;i<N;i++)
  {
    // alege x
    u = gsl_rng_uniform( r);
    x = a + (b-a) * u;
    // alege y
    u = gsl_rng_uniform( r);
    y = c * u;
    // verifica f(x)<y
    if ( y < f(x) )
       {
       N1 = N1 + 1;
       }
    else
       {
       N2 = N2 + 1;
       }
  }
  // valoarea integralei
  I = (double)N1/(double)N * (b-a)*c;

  return I;
}

int main( int argc, char **argv)
{
int Np, rank;
long i;
// limitele de integrat
double a, b, c;
double N;
double c1, c2;
long Ni;

MPI_Init( &argc, &argv);
MPI_Comm_size( MPI_COMM_WORLD, &Np);
MPI_Comm_rank( MPI_COMM_WORLD, &rank);

// MASTER -- rang 0
```

```
if (rank==0)
{
    double s, I;
    // input
    a = 0.;
    b = 1.;
    c = 4.;
    N = 100000000;
    // trimite a, b, c catre procesele de tip worker
    for (i=1;i<Np;i++)
    {
         // trimite intervalul de integrare [a,b]
         MPI_Send( &a, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
         MPI_Send( &b, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
         // trimite valoarea maxima c
         MPI_Send( &c, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
         // trimite numarul de puncte
         Ni = (long) ( (double)N/(double)(Np-1) );
         MPI_Send( &Ni, 1, MPI_LONG, i, 0, MPI_COMM_WORLD);
    }
    // primeste rezultatele integralelor si calculeaza
       rezultatul final
    I = 0.;
    for (i=1;i<Np;i++)
    {
         MPI_Recv( &s, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
            MPI_COMM_WORLD,
                   MPI_STATUS_IGNORE);
         I = I + s;
    }
    I = I / (double)(Np-1);
    // afiseaza rezultatul integralei
    printf( "I = %f \n", I);
}

// WORKERS -- rangurile 1, 2, ... ,
else
{
double s;
// primeste intervalul de integrare [a,b]
MPI_Recv( &a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
```

```
MPI_Recv( &b, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
// primeste valoarea maxima c
MPI_Recv( &c, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
// primeste numarul de puncte Ni
MPI_Recv( &Ni, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
printf( "%f %f %f %i\n", a, b, c, Ni);
// calculeaza integrala
s = calculeaza_integrala( a, b, c, Ni);
printf("s=%f\n", s);
// trimite rezultatul catre MASTER
MPI_Send( &s, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}

MPI_Finalize();

return 0;
}
```

Capitolul 4

Operaţii de comunicare colective

MPI pune la dispoziţie operaţii de comunicare colective [2, 3]. Spre deosebire de operaţiile de tip point-to-point menţionate anterior, în cadrul unei operaţii colective sunt implicate toate procesele din cadrul unui comunicator. În cazul în care se doreşte să se efectueze o operaţie colectivă pe un anumit subgrup de procese din comunicator, este necesară definirea unui nou comunicator.

Operaţiile de comunicare globală nu pot interfera cu cele de tip point-to-point, fiind apelate de către fiecare rang, iar procesele implicate pot fi sincronizate sau nu în urma aplicării operaţiei.

Există similarităţi cu operaţiile de tip point-to-point, dar şi unele diferenţe. La ambele tipuri de operaţii, datele se transmit sub forma unei secvenţe, specificând tipul de date. În cadrul operaţiilor de comunicare globală nu există însă noţiunea de etichetă.

4.1 Operaţii de tip broadcast, scatter, gather

4.1.1 `MPI_Bcast`

Operaţia de *single broadcast*, definită de funcţia `MPI_Bcast()`, trimite un bloc de date x de la un proces, către toate procesele din comunicator. Rangul care trimite blocul de date x se numeşte proces *root*.

Înaintea execuţiei operaţiei de broadcast, mesajul este stocat în memoria locală a procesorului root, iar după execuţie mesajul va fi stocat în memoria locală a fiecărui procesor. Pentru a efectua operaţia, fiecare procesor trebuie să execute o operaţie de tip broadcast în care se specifică procesorul root.

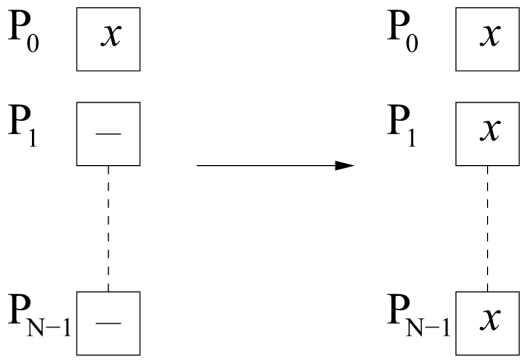

Figura 4.1: MPI single broadcast. Datele x sunt distribuite de la procesul P_0 (nodul root) către toate procesele.

```
int MPI_Bcast (void *message, int count, MPI_Datatype type,
               int root, MPI_Comm comm);
```

- `root`: reprezintă procesul care trimite date;
- `message`: bufferul send (procesul root) / receive (celelalte procese)

Observaţii:

- fiecare proces cheamă operaţia `MPI_Bcast`, specificând acelaşi root şi acelaşi comunicator;
- datele trimise prin `MPI_Bcast()` nu pot fi recepţionate folosind `MPI_Recv()`;

Exemplu de cod `MPI_Bcast()`:

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main( int argc, char **argv)
{
```

```
int i;
int numprocs, rank;
int x;
int root = 1;

// initializare MPI
MPI_Init( &argc, &argv);
MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank( MPI_COMM_WORLD, &rank);

x = rank + 1;

MPI_Bcast( &x, 1, MPI_INT, root, MPI_COMM_WORLD);

printf( "Rank %i, x = %i \n", rank, x);

MPI_Finalize();

return 0;
}
```

Variabila x, care reprezintă blocul de date ce urmează a fi distribuit, este iniţializată diferit pentru fiecare proces, cu valoarea $rang + 1$. Este selectat nodul *root* ca fiind procesul cu rangul 1. Se aplică operaţia *broadcast*, care are ca efect modificarea variabilei x în fiecare proces, cu valoarea iniţializată în rangul 1 ($x = 2$). Făcând abstracţie de ordinea apariţiei rangurilor, codul va avea ca output:

Rank 1, x = 2
Rank 0, x = 2
Rank 2, x = 2
Rank 3, x = 2

4.1.2 MPI_Scatter

Apelând operaţia de comunicare globală `MPI_Scatter` procesul root trimite câte un bloc de date către toate procesele.

```
int MPI_Scatter( void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
```

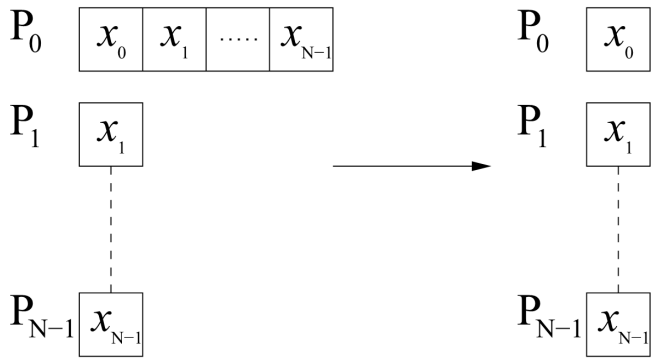

Figura 4.2: Operaţia `MPI_Scatter()`. Datele sunt distribuite de la nodul *root* către toate procesele.

```
                int root, MPI_Comm comm);
```

- `sendbuf`: bufferul send pus la dispoziţie de procesul root, care conţine câte un bloc pentru fiecare proces;
- `sendcount`, `sendtype`: numărul de elemente, tipul lor;
- `recvbuf`, `recvcount`, `recvtype`: buffere specificate de către fiecare proces, numărul de elemente primite, tipul elementelor;

Fiecare proces trebuie să specifice acelaşi root, număr de elemente şi acelaşi tip !

Exemplu de cod `MPI_Scatter()`:

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main( int argc, char **argv)
{

int i;
int numprocs, rank;
int *sbuf;
int *rbuf;
int Nr = 10;
```

```
int Ns;
int root = 0;

// initializare MPI
MPI_Init( &argc, &argv);
MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank( MPI_COMM_WORLD, &rank);

Ns = Nr * numprocs;

// alocare memorie
if (rank==root) sbuf = malloc( Ns*sizeof(int));
rbuf = malloc( Nr*sizeof(int));

if (rank==root) {
    for (i=0;i<Ns;i++) {
         sbuf[i] = i + 1;
    }
}

MPI_Scatter( sbuf, Nr, MPI_INT, rbuf, Nr, MPI_INT, root,
   MPI_COMM_WORLD);

for (i=0;i<Nr;i++) {
     printf( "rank %i, rbuf[%i] = %i \n", rank, i, rbuf[i]);
}


MPI_Finalize();

return 0;
}
```

4.1.3 MPI_Gather

Operaţia Gather:
- fiecare procesor pune la dispoziţie un bloc, care apoi este colectat la procesorul root;
- spre deosebire de `MPI_Reduce()` nu se efectuează operaţia de reducere;

```
int MPI_Gather( void *sendbuf, int sendcount, MPI_Datatype sendtype,
```

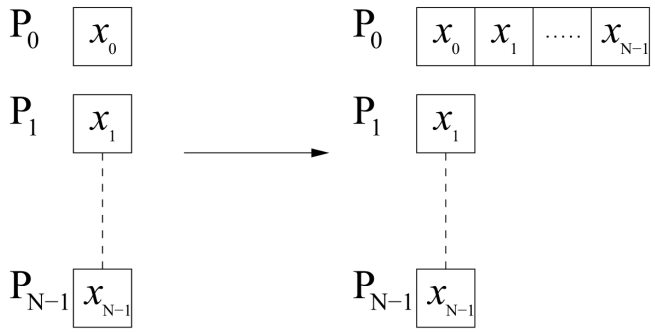

Figura 4.3: Operaţia `MPI_Gather()`. Datele sunt preluate de la toate procesele către nodul *root*.

```
void *recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm);
```

- `sendbuf`: se specifică buferul send de către fiecare proces;
- `sendcount`, `sendtype`: numărul de elemente, tipul lor;
- `recvbuf`: bufferul în care se primesc datele, la procesul root;
- `recvcount`, `recvtype`: numărul de elemente şi tipul lor;

Exemplu de cod `MPI_Gather()`:

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main( int argc, char **argv)
{

int i;
int numprocs, rank;
int *sbuf;
int *rbuf;
int N = 10; // N elemente in fiecare proces
int root = 0;

// initializare MPI
```

```
MPI_Init( &argc, &argv);
MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank( MPI_COMM_WORLD, &rank);

// alocare memorie
sbuf = malloc( N*sizeof(int));
if (rank==root) rbuf = malloc( N*numprocs*sizeof(int));

for (i=0;i<N;i++) {
     sbuf[i] = rank*100 + i;
}

MPI_Gather( sbuf, N, MPI_INT, rbuf, N, MPI_INT, root,
   MPI_COMM_WORLD);

if (rank==0) {
    for (i=0;i<N*numprocs;i++) {
         printf( "rbuf[%i] = %i \n", i, rbuf[i]);
    }
}

MPI_Finalize();

return 0;
}
```

4.1.4 Număr diferit de elemente - `MPI_Scatterv`, `MPI_Gatherv`

Operaţiile de comunicare colective `MPI_Scatter` şi `MPI_Gather` descrise anterior privesc transmiterea unor segmente de date de acelaşi tip şi aceeaşi mărime. Pentru mai multă flexibilitate, se pot folosi funcţiile `MPI_Scatterv` şi `MPI_Gatherv`, ale căror prototipuri sunt indicate mai jos.

```
int MPI_Scatterv( void *sendbuf, int *sendcounts, int *displs,
              MPI_Datatype sendtype, void *recvbuf, int recvcount,
              MPI_Datatype recvtype, int root, MPI_Comm comm);

int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
              void *recvbuf, const int *recvcounts, const int *displs,
```

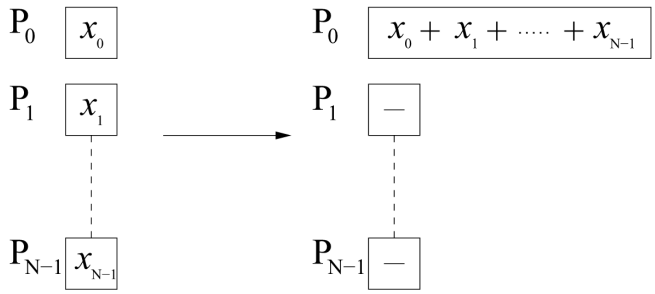

Figura 4.4: Operaţia MPI de reducere globală. Datele sunt preluate de la toate procesele către nodul *root* unde se efectuează operaţia de reducere.

```
                MPI_Datatype recvtype, int root, MPI_Comm comm);
```

Acestea diferă de operaţiile standard prin următoarele două modificări:

- `*sendcounts`: vector cu numărul de elemente trimise către fiecare proces;
- `*displs`: specifică poziţia blocurilor de elemente din bufferul send;

în cazul `MPI_Scatterv()` şi

- `revcounts`: vector cu numărul de elemente primite de la fiecare proces;
- `*displs`: specifică poziţia blocurilor de elemente din bufferul recvbuf;

în cazul `MPI_Gatherv()`.

4.2 Operaţii de reducere globală – MPI_Reduce

În cadrul operaţiei de reducere globală, `MPI_Reduce()` fiecare procesor pune la dispoziţie un bloc de date de acelaşi tip şi aceeaşi mărime. Datele sunt adunate la nodul *root* şi se efectuează operaţia de reducere. Pentru a efectua operaţia de single accumulation fiecare nod cheamă operaţia specificând nodul root.

Exemplu de cod `MPI_Reduce()`:

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <mpi.h>

int main( int argc, char **argv)
{

int i;
int numprocs, rank;
int *sbuf;
int *rbuf;
int N = 10; // N elemente in fiecare proces
int root = 0;

// initializare MPI
MPI_Init( &argc, &argv);
MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank( MPI_COMM_WORLD, &rank);

// alocare memorie
sbuf = malloc( N*sizeof(int));
rbuf = malloc( N*sizeof(int));

for (i=0;i<N;i++) {
     sbuf[i] = rank*100 + i;
}

MPI_Reduce( sbuf, rbuf, N, MPI_INT, MPI_SUM, root,
   MPI_COMM_WORLD);

if (rank==0) {
    for (i=0;i<N;i++) {
         printf( "Element %i, suma = %i \n", i, rbuf[i]);
    }
}

MPI_Finalize();

return 0;
}
```

MPI pune la dispoziţie un număr de operaţii de reducere, care sunt operaţii comutative.

Sintaxa MPI	Operaţie
MPI_MAX	maxim
MPI_MIN	minim
MPI_SUM	suma
MPI_LAND	şi logic
MPI_BAND	şi binar
MPI_LOR	sau logic
MPI_BOR	sau binar
MPI_LXOR	sau exclusiv logic
MPI_BXOR	sau exclusiv binar
MPI_MAXLOC	maxim şi locaţia în şir
MPI_MINLOC	minim şi locaţia în şir

Tabela 4.1: Operaţii de reducere

Pe lângă aceste operaţii predefinite se pot defini funcţii de reducere noi, folosind `MPI_Op_create`:

```
int MPI_Op_create( MPI_User_function *function, int commute,
                   MPI_Op *op);
```

Observaţii:

- `function` trebuie să aibe următoarele 4 variabile: `void *in, void *out, int *len, MPI_Datatype *type;`
- funcţia trebuie să fie asociativă;
- variabila `commute` specifică dacă funcţia este comutativă (commute=1) sau nu (commute=0);
- funcţia `MPI_Op_create` returnează o operaţie de reducere `op` care poate fi folosită ca parametru în `MPI_Reduce()`;

4.3 Multi-broadcast

Operaţia de tip Multi-broadcast:
- fiecare proces pune la dispoziţie un bloc;

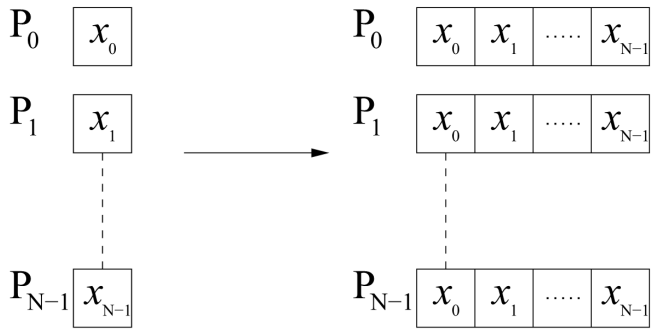

Figura 4.5: Multi-broadcast - Operaţia `MPI_Allgather()`.

- după executarea operaţiei, fiecare bloc va fi trimis către toate procesele din comunicator;

```
int MPI_Allgather( void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype, MPI_Comm comm);

int MPI_Allgatherv( void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int *recvcounts, int *displs,
               MPI Datatype recvtype, MPI_Comm comm);
```

4.4 Multi-accumulation

Operaţia de tip multi-accumulation:
- fiecare proces execută o operaţie de tip single-accumulation;
- rezultatul este echivalent cu o operaţie de single-accumulation, urmată de single-broadcast;

```
int MPI_Allreduce( void *sendbuf, void *recvbuf, int count,
               MPI_Datatype type, MPI_Op op, MPI_Comm comm);
```

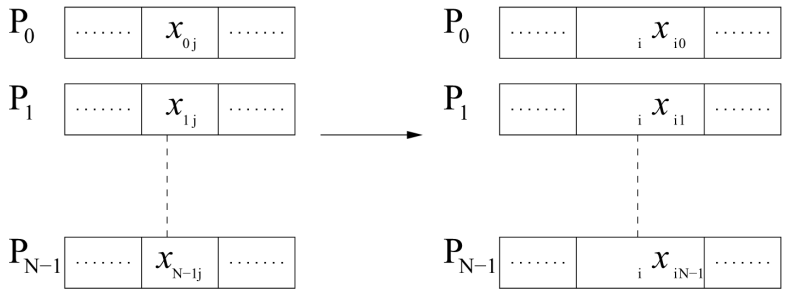

Figura 4.6: Multi-accumulation - Operaţia `MPI_Allreduce()`.

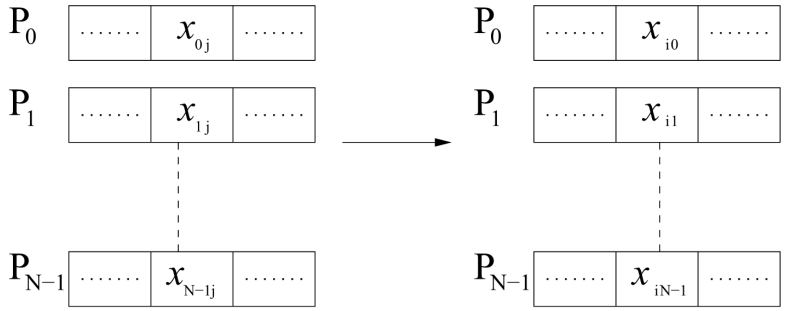

Figura 4.7: Total exchange - Operaţia `MPI_Alltoall()`.

4.5 Total exchange

Operaţia de tip total-exchange:
- fiecare proces pune la dispoziţie un bloc de date diferit pentru fiecare alt proces;
- operaţia este echivalentă cu următoarele operaţii:
  - fiecare proces execută o operaţie scatter (sender view);
  - fiecare proces execută o operaţie gather (receiver view);

```
int MPI_Alltoall( void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, MPI_Comm comm);
```

Pentru blocuri de dimensiuni diferite există versiunea `MPI_Alltoallv()`.

Capitolul 5

Comunicatori şi grupuri

Transmiterea mesajelor între diferitele noduri care participă la execuţia unui program MPI în paralel poate fi o operaţie costisitoare în termeni de timp de calcul şi resurse. Regula de bază în programarea MPI rămâne reducerea numărului de mesaje transmise între procese: cu cât mai puţine mesaje transmise, cu atât programul este mai performant.

Din acest motiv standardul MPI conţine mecanisme atât pentru gruparea unor seturi de date individiuale într-un singur mesaj (tipuri de date derivate, funcţiile de tip `MPI_Pack()/MPI_Unpack()`), dar şi pentru definirea unor grupuri de procese pentru facilitarea schimbului de date [2]. Un *grup* este o mulţime ordonată de procese, fiecare proces fiind identificat în mod unic prin rangul său. Dacă grupul conţine np procese, rangurile lor iau valorile de la 0 la $np - 1$.

Un *comunicator* este definit de un *grup* de procese şi de un *context*. Contextul este un obiect (dependent de sistem)[1] care identifică în mod unic un comunicator (reciproca este de asemenea adevărată: un comunicator are un unic context). Doi comunicatori diferiţi au contexte distincte, dar pot avea acelaşi grup de procese.

La lansarea în execuţie a unui program în paralel, biblioteca MPI creează un comunicator implicit, `MPI_COMM_WORLD`, al cărui grup conţine toate procesele iniţiale (care pot schimba mesaje între ele sau pot participa la operaţii colective de transmitere de date, în contextul asociat acestui comunicator).

Crearea de noi comunicatori poate fi necesară pentru:

- asigurarea operaţiilor colective de transmitere de date în cadrul unei submulţimi

[1] Grupurile şi comunicatorii sunt obiecte opace, în sensul că detaliile implementării lor nu sunt accesibile direct utilizatorului. Acesta are acces la ele prin intermediul unui *handle* şi a unor funcţii MPI special create pentru a le manipula.

a proceselor iniţiale;

- crearea unei topologii virtuale a proceselor, pentru a exploata la maximum topologia reală a clusterului pe care este executat programul;
- asigurarea modularităţii necesare atunci când aplicaţia (programul) apelează funcţii dintr-o bibliotecă care la rândul lor sunt implementate pe baza standardului MPI (separarea contextului aplicaţiei de contextul necesar execuţiei funcţiilor din biblioteca externă).

5.1 Funcţii MPI pentru operaţii cu grupuri şi comunicatori

În această secţiune sunt prezentate câteva dintre funcţiile utile pentru definirea de noi grupuri şi comunicatori, definite de standardul MPI. De asemenea, cu titlu de exemplu de utilizare, sunt indicate două programe MPI. Funcţionalitatea acestor programe este suficientă rolului lor ilustrativ.

```
int MPI_Comm_group( MPI_Comm comm, MPI_Group *group)
```

Scop: întoarce grupul asociat comunicatorului `comm`.
Parametrii de intrare: comunicatorul `comm`
Parametrii de ieşire: handle pentru grupul comunicatorului (pointerul `group`) şi un cod de eroare ca valoare de întoarcere a funcţiei.

```
int MPI_Group_incl( MPI_Group group, int n, int *ranks,
                MPI_Group *newgroup)
```

Scop: defineşte un nou grup prin reordonarea unui grup existent şi includerea doar a membrilor indicaţi.
Parametrii de intrare: grupul iniţial (handle) `group`; dimensiunea noului grup `n` (valoare întreagă); rangurile proceselor din grupul iniţial `group` care vor fi incluse în noul grup (şir de `n` valori întregi, reperat de pointerul `ranks`).
Parametrii de ieşire: handle pentru noul grup (pointerul `newgroup`) care conţine procesele în ordinea definită de `ranks` (altfel spus, procesul cu rangul `k` în noul grup `newgroup` este procesul cu rangul `rank[k]` din `group`) şi un cod de eroare ca valoare de întoarcere a funcţiei.

```
int MPI_Comm_create( MPI_Comm comm, MPI_Group group,
                MPI_Comm *newcomm)
```

Scop: creează un nou comunicator, definit de grupul `group` şi un nou context; toate procesele din `comm` trebuie să apeleze funcţia, pentru a defini corect contextul noului comunicator.
Parametrii de intrare: comunicatorul iniţial (handle) `comm`; grupul `group` asociat noului comunicator (submulţime a grupului asociat comunicatorului iniţial).
Parametrii de ieşire: handle pentru noul comunicator (pointerul `newcomm`) şi un cod de eroare ca valoare de întoarcere a funcţiei.

```
int MPI_Comm_split( MPI_Comm comm, int color, int key,
                MPI_Comm *newcomm)
```

Scop: creează noi comunicatori pe baza unor "culori" şi a unor coduri sau "chei". Această funcţie creează o partiţie a grupului asociat comunicatorului `comm` în subgrupuri disjuncte, câte un subgrup pentru fiecare valoare `color`. Fiecare subgrup conţine procesele cu aceeaşi culoare. În interiorul fiecărui subgrup procesele sunt ordonate după valoarea parametrului `key`. Este creat un nou comunicator pentru fiecare subgrup. Toate procesele din comunicatorul `comm` trebuie să apeleze funcţia, pentru a defini corect contextul noului comunicator.
Parametrii de intrare: comunicatorul iniţial (handle) `comm`; parametrul de control pentru definirea subgrupurilor `color` (întreg pozitiv sau zero); parametrul de control pentru atribuirea rangurilor `key`.
Parametrii de ieşire: handle pentru noul comunicator (pointerul `newcomm`) şi un cod de eroare ca valoare de întoarcere a funcţiei.

Exemplele următoare ilustrează modalitatea de definire a unui nou grup de procese pe baza grupului comunicatorului implicit şi crearea unui nou comunicator. Primul utilizează funcţia `MPI_Comm_create()`, iar al doilea funcţia `MPI_Comm_split()`. Liniile de cod sunt comentate explicit.

Exemplul 1:

```
/*************************************************************
```

```
 Programul separa procesele din grupul comunicatorului initial
      in doua subgrupuri distincte, pe baza rangului lor, dupa
      care sunt creati comunicatorii corespunzatori. Apoi sunt
      efectuate operatii de comunicare colective in interiorul
      acestor comunicatori. Scopul este unul pur demonstrativ.
*************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

// fixam numarul de procese, pentru a asigura functionalitatea
     corecta
// a programului (acest program are exclusiv un scop
    ilustrativ)
#define NUMPROCS 6

int main(int argc, char **argv)
{
  int myRank, myNewRank, nProc; // rangul procesului curent,
      in comunicatorul initial si in cel derivat, si numarul
      de procese active
  int sendBuf, recvBuf;          // bufere pentru operatiile de
       comunicare colective


  int evenRanks[3] = {0,2,4};    // pentru definirea
      subgrupului proceselor cu rang initial par
  int oddRanks[3] = {1,3,5};    // pentru definirea subgrupului
       proceselor cu rang initial impar

  MPI_Group initial_group, new_group;    // handlere pentru
      grupul comunicatorului initial si pentru noul grup
  MPI_Comm initial_comm = MPI_COMM_WORLD, new_comm;    //
      handlere pentru comunicatorul initial si pentru noul
      comunicator



  MPI_Init(&argc,&argv);  // initializarea mediului MPI; numai
       dupa acest pas pot fi apelate functiile MPI
```

```
MPI_Comm_rank(initial_comm, &myRank);  // rangul procesului
    curent, in comunicatorul implicit
MPI_Comm_size(initial_comm, &nProc);  // numarul de procese
    active

// Pentru a asigura functionalitatea programului, fortam
    lansarea in executie a NUMPROCS procese,
// in caz contrar terminand sesiunea MPI curenta
if ( nProc != NUMPROCS )
  {
    if (myRank == 0)
      printf("Programul functioneaza corect numai daca sunt
          lansate in executie %d procese. Ies...\n",
          NUMPROCS);
    MPI_Finalize();           // terminam sesiunea MPI
        curenta inainte de iesire
    exit(1);                  // si iesim cu cod 1
  }

// stocam rangul initial in sendBuf
sendBuf = myRank;
// apelam MPI_Comm_group(), pentru a obtine un handle pentru
    grupul initial
MPI_Comm_group(initial_comm, &initial_group);

// cream cele doua grupuri de procese distincte, pe baza
    valorii rangului initial (par sau impar)
if (myRank % 2)
  MPI_Group_incl(initial_group, 3, oddRanks, &new_group);
     // subgrupul proceselor cu rang impar
else
  MPI_Group_incl(initial_group, 3, evenRanks, &new_group);
     // subgrupul proceselor cu rang par

 // cream noul comunicator, caruia ii este asociat new_group
MPI_Comm_create(initial_comm, new_group, &new_comm);  //
    new_comm contine grupuri distincte pe procesele cu rang
    par/impar

 // rangul procesului curent in noul grup
MPI_Group_rank(new_group, &myNewRank);
```

```
  // operatie colectiva: se aduna valorile sendBuf din toate
     procesele dintr-un
  // grup si se intoarce rezultatul in recvBuf de pe procesele
      cu rangul nou 0
  MPI_Reduce(&sendBuf, &recvBuf, 1, MPI_INT, MPI_SUM, 0,
     new_comm);

  // si afisam cateva informatii, cu rol demonstrativ (se
     sumeaza valorile
  // sendBuf din toate procesele dintr-un grup si se intoarce
     rezultatul in
  // recvBuf de pe procesele cu rangul nou 0 - acum doua
     asemenea procese)
  if (myNewRank == 0)
    printf("Procesul cu rangul initial = %d, noul rang = %d,
       are recvBuf = %d\n", myRank, myNewRank, recvBuf);

  // terminam sesiunea MPI
  MPI_Finalize();
  // ...si iesim
  return 0;

}
```

Exemplul 2:

```
/*****************************************************
Programul demonstreaza modul de creare a unui set de n
   comunicatori cu utilizarea functiei MPI_Comm_split().
   Forma actuala presupune ca numarul de procese lansate in
   executie este np = n*n. Dupa crearea noilor comunicatori
   se executa o operatie colectiva de transmitere de date.
   Cod adaptat dupa P. Pacheco, "Parallel Programming with
   MPI" np = n * n procese distribuite pe un grid de n x n
   noduri
 * Exemplu (n=2):
 *                      ---------
 *            linia 0   | 0 | 1 |
 *                      ---------
 *            linia 1   | 2 | 3 |
 *                      ---------
```

```
 *
 Se defineste un comunicator pentru fiecare linie, adica
    pentru procesele {0,1} si {2,3}, pentru a facilita
    comunicarea directa intre ele.
*****************************************************/

#include <stdio.h>
#include <math.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char **argv)
{
  int nProc, myRank;              // nr. de procese in executie
     si rangul
  MPI_Comm my_row_comm;           // comunicatorul
     corespunzatori unei linii
                                  // in topografia nodurilor de
                                     calcul

  int my_row, my_rank_in_row;     // linia si rangul unui proces
      in linie
  int n;                          // variabila interna; n*n =
     nProc
  char test[32];                  // variabila interna, pentru
     operatii colective

  MPI_Init(&argc, &argv);         // se initializeaza mediul MPI

  // nProc va contine numarul de procese lansate in executie
     in comunicatorul
  // implicit MPI_COMM_WORLD
  MPI_Comm_size(MPI_COMM_WORLD, &nProc);

  // myRank va contine rangul procesului curent
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  // programul presupune o topologie 2D n x n a nodurilor de
     calcul pe
  // care sunt distribuite cele nProc procese lansate in
     executie
  // determinam numarul de linii n
```

```
  n = (int) sqrt((double) nProc);

  // numarul liniei corespunzatoare procesului curent este
     stocat in my_row
  my_row = myRank/n;
  MPI_Comm_split(MPI_COMM_WORLD, my_row, myRank,
                          &my_row_comm);

  // si testam noii comunicatori; mai intai, aflam noile
     ranguri ale proceselor
  MPI_Comm_rank(my_row_comm, &my_rank_in_row);

  // procesele cu rang zero in fiecare linie definesc stringul
      test
  if (my_rank_in_row == 0)
    strcpy(test,"Test reusit!");

  // si il transmitem colectiv tuturor proceselor din noii
     comunicatori
  MPI_Bcast(test, 32, MPI_CHAR, 0, my_row_comm);

  // si verificam rezultatul (cu scop exclusiv demonstrativ!)
  printf("Procesul cu rangul initial %d : linia = %d, noul
     rang = %d, "
         "stringul test = %s\n", myRank, my_row,
            my_rank_in_row, test);

  MPI_Finalize();        // terminam sesiunea MPI

  return 0;              // ...si iesim
}
```

Capitolul 6

Operaţii I/O în MPI

Operaţiile I/O (Input/Output) în prima versiune a standardului MPI vizau de regulă utilizarea unui singur nod, cu acces la mediile de stocare sau la cele de intrare. Acest tip de abordare devine ineficient în momentul în care cantităţile de date implicate în operaţiile I/O sunt mari, datorită supraîncărcării reţelei de transmitere de date şi implicit a performanţei programului.

Standardul MPI 2.0 propune o soluţie a acestei probleme sub forma unor funcţii dedicate pentru operaţii I/O desfăşurate în paralel, majoritatea acestor funcţii fiind astăzi disponibile în principalele biblioteci MPI.

Pentru început, o precizare: numim operaţie I/O desfăşurată în paralel (sau, mai pe scurt, paralelă) acea operaţie în care mai multe procese ale unui program paralel accesează date (în regim de intrare sau ieşire) din *acelaşi* fişier.

6.1 Funcţii I/O definite de standardul MPI 2.0

Această secţiune este dedicată prezentării unora dintre cele mai des folosite funcţii MPI de acces colectiv la fişiere.

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode,
                  MPI_Info info, MPI_File *fp)
```

Scop: deschide un fişier pentru acces colectiv

Parametrii de intrare:
- `comm` - comunicatorul (handle) în al cărui context este deschis fişierul. Toate procesele din acest comunicator deschid acest fişier. Dacă se doreşte deschiderea fişierului pe un singur proces, independent de celelalte, se va utiliza valoarea `MPI_COMM_SELF`;
- `filename` - numele fişierului către care se deschide accesul (string);
- `amode` - modul de acces la fişier (valoare întreagă). Sunt suportate următoarele valori predefinite pentru modul de acces:
`MPI_MODE_RDONLY` - fişierul este deschis numai pentru citire de date
`MPI_MODE_RDWR` - fişierul este deschis pentru operaţii de citire/scriere
`MPI_MODE_WRONLY` - fişierul este deschis/creat doar pentru scriere
`MPI_MODE_CREATE` - fişierul este creat, dacă nu există
`MPI_MODE_EXCL` - generează un cod de eroare dacă se încearcă crearea unui fişier care există deja
`MPI_MODE_DELETE_ON_CLOSE` - şterge fişierul creat la închidere (pentru fişiere temporare)
`MPI_MODE_UNIQUE_OPEN` - fişierul nu va fi deschis simultan în altă parte
`MPI_MODE_SEQUENTIAL` - fişierul va fi accesat doar în mod secvenţial de către procesele care-l deschid
`MPI_MODE_APPEND` - fişierul este deschis în mod adăugare (poziţia iniţială a pointerilor de fişier este fixată la sfârşitul fişierului).
Pot fi specificate mai multe moduri de acces, în forma: `MPI_MODE_WRONLY | MPI_MODE_DELETE_ON_CLOSE`.
- `info` - obiect de tip MPI_Info (handle), prin intermediul căruia utilizatorul poate transmite informaţii de tipul:
`MPI_INFO_NULL` - nici o informaţie
`shared_file_timeout` - durata (în secunde) aşteptării pentru accesul la pointerul de fişier, înainte de ieşirea din funcţie cu codul `MPI_ERR_TIMEDOUT`
`rwlock_timeout` - durata (în secunde) aşteptării pentru a obţine blocarea (*lock*) operaţiilor I/O asupra unei porţiuni contigue (simplu conexe) a fişierului, înainte de ieşirea din funcţie cu codul `MPI_ERR_TIMEDOUT`
`noncoll_read_bufsize` - dimensiunea maximă a zonei tampon (*buffer*) utilizată de mediul MPI I/O pentru a satisface cereri multiple de citire noncontiguă în rutinele de acces de date în mod non-colectiv
`noncoll_write_bufsize` - dimensiunea maximă a zonei tampon (*buffer*) utilizată de

mediul MPI I/O pentru a satisface cereri multiple de scriere noncontiguă în rutinele de acces de date în mod non-colectiv
`coll_read_bufsize` - dimensiunea maximă a zonei tampon (*buffer*) utilizată de mediul MPI I/O pentru a satisface cereri multiple de citire noncontiguă în rutinele de acces de date în mod colectiv
`coll_write_bufsize` - dimensiunea maximă a zonei tampon (*buffer*) utilizată de mediul MPI I/O pentru a satisface cereri multiple de scriere noncontiguă în rutinele de acces de date în mod colectiv.
Parametrii de ieşire:
`fp` - noul descriptor de fişier (handle).
Utilizatorul este responsabil pentru închiderea tuturor fişierelor deschise cu `MPI_File_open()` înainte de terminarea sesiunii MPI prin apelul funcţiei `MPI_Finalize()`.

```
int MPI_File_close(MPI_File *fp)
```

Scop: închide un fişier deschis anterior pentru acces colectiv. Fişierul este şters dacă a fost deschis în modul `MPI_MODE_DELETE_ON_CLOSE`. Utilizatorul este responsabil pentru asigurarea încheierii tuturor operaţiilor de acces fără blocare (non-blocking), înainte de apelul acestei funcţii. Pointerul `fp` este redefinit la valoarea `MPI_FILE_NULL`.

```
int MPI_File_read(MPI_File fh, void *buf, int count,
                  MPI_Datatype datatype, MPI_Status *status)
```

Scop: citeşte date dintr-un fişier de la poziţia specificată de pointerul de fişier individual. MPI menţine un pointer de fişier individual pentru fiecare proces, pentru fiecare descriptor de fişier. Este o functţie de tip blocking (cu blocare), non-colectivă.
Parametrii de intrare:
- `fh` - descriptorul de fişier (handle);
- `count` - numărul de elemente de tip `datatype` care sunt citite (extrase) din fişier;
- `datatype` - tipul de date citite.
Parametrii de ieşire:
- `buf` - adresa iniţială a spaţiului RAM alocat pentru datele citite;
- `status` - obiect pentru identificarea stării operaţiei (de exemplu, numărul de elemente de tip `datatype` citite). Dacă nu este folosit, se va indica valoarea `MPI_STATUS_IGNORE`.

Această funcţie nu va fi apelată dacă a fost specificat modul de acces `MPI_MODE_SEQUENTIAL` la deschiderea fişierului.

```
int MPI_File_read_all(MPI_File fh, void *buf, int count,
                MPI_Datatype datatype, MPI_Status *status)
```

Scop: citeşte date dintr-un fişier de la poziţia specificată de pointerii de fişier individuali, cu acces colectiv. Funcţia este de tip blocking (cu blocare). Este varianta colectivă a funcţiei `MPI_File_read()` descrisă anterior, utilă dacă toate procesele lansate în execuţie trebuie să citească simultan date din fişier, la un moment dat. Semnificaţia parametrilor acestei funcţii este aceeaşi ca în cazul funcţiei `MPI_File_read()`.

```
int MPI_File_write(MPI_File fh, void *buf, int count,
                MPI_Datatype datatype, MPI_Status *status)
```

Scop: scrie date într-un fişier, în locul specificat de pointerul de fişier individual. MPI menţine un pointer de fişier individual pentru fiecare proces, pentru fiecare descriptor de fişier. Funcţiile de tip `MPI_File_read(), MPI_File_write()` actualizează valoarea pointerului individual.
Parametrii de intrare:
- `fh` - descriptorul de fişier (handle);
- `buf` - adresa de început a zonei de memorie (RAM) unde sunt stocate datele care urmează a fi scrise în fişier;
- `count` - numărul de elemente de tip `datatype` care urmează a fi scrise;
- `datatype` - tipul de date al fiecărui element;
Parametrii de ieşire:
- `status` - obiect pentru identificarea stării operaţiei (de exemplu, numărul de elemente de tip `datatype` scrise). Dacă nu este folosit, se va indica valoarea `MPI_STATUS_IGNORE`.

```
int MPI_File_write_all(MPI_File fh, void *buf, int count,
                MPI_Datatype datatype, MPI_Status *status)
```

Scop: scrie date într-un fişier la poziţia specificată de pointerii de fişier individuali, cu acces colectiv. Funcţia este de tip blocking (cu blocare). Este varianta colectivă a funcţiei `MPI_File_write()`, descrisă mai sus. Semnificaţia parametrilor este aceeaşi ca în cazul funcţiei `MPI_File_write()`.

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf,
                int count, MPI_Datatype datatype, MPI_Status *status)
```

Scop: scrie date într-un fişier în locul indicat de parametrul `offset`. Semnificaţia celorlalţi parametri este aceea indicată în cazul funcţiei `MPI_File_write()`. Este o funcţie cu blocare, non-colectivă. Versiunea ei colectivă este:
`int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`.

```
int MPI_File_write_ordered(MPI_File fh, void *buf, int count,
                MPI_Datatype datatype, MPI_Status *status)
```

Scop: scrie date într-un fişier, în locul specificat de un pointer de fişier comun tuturor proceselor. Este o funcţie cu blocare, de tip colectiv. Semnificaţia parametrilor este aceea indicată în cazul funcţiei

```
MPI_File_write()
```

Ordinea de acces la fişier este determinată de rangul proceselor din grup. Funcţia trebuie apelată de toate procesele din grupul în care este definit `fh`. Fiecare proces poate însă transmite valori diferite pentru argumentele `datatype` şi `count`. Pointerul de fişier comun este modificat după fiecare operaţie de scriere.

```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
```

Scop: întoarce dimensiunea curentă a fişierului identificat de `fh`, în octeţi, în variabila `size`.

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
                MPI_Datatype filetype, char *datarep, MPI_Info info)
```

Scop: modifică pointerul de fişier, astfel încât operaţiile I/O ulterioare să acceseze porţiunea corectă din fişier.
Parametrii de intrare:
- `disp` - noul offset al pointerului de fişier;
- `etype` - tipul elementar de date;
- `filetype` - tipul de date (derivat, pentru un control mai bun al operaţiei); în cazuri simple, este acelaşi ca `edat`;

- `datarep` - reprezentarea datelor (string), introdus din motive tehnice; valoarea `native` este de regulă suficientă;
- `info` - obiect de tip info (`MPI_INFO_NULL` dacă nu e folosit).

Această funcţie resetează pointerii de fişier individuali, dar şi pointerul colectiv de fişier la zero. Este colectivă: toate procesele din grup trebuie să o apeleze cu aceeaşi valoare `datarep` şi cu valori `etype` de aceeaşi lungime. Valorile parametrilor `disp, filetype, info` pot fi diferite.

6.2 Exemple de utilizare a funcţiilor MPI pentru operaţii I/O

Exemplele următoare demonstrează modul de utilizare a unora dintre funcţiile MPI indicate în secţiunea 6.1 pentru citire de date dintr-un fisier binar (`MPI_File_read()`), respectiv pentru scriere de date în paralel într-un fişier binar (`MPI_File_write_at()`). Liniile de cod sunt comentate explicit.

1. Accesarea în paralel a unui fişier de date pentru citire, cu funcţiile MPI

```
/*************************************************
 * Programul demonstreaza operatia de citire paralela,
   cu un numar arbitrar de procese
*************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"


int main(int argc, char **argv)
{
  int myRank, nProc;     // rangul procesului curent si numarul de
      procese
  int dataSize, dataElements;   // variabile interne
  float *data;           // adresa RAM unde sunt stocate datele

  MPI_File fh;           // descriptorul de fisier
  MPI_Status status;     // pentru starea operatiei
```

```
MPI_Offset fileSize;
              // pastreaza dimensiunea fisierului
char fname[32];        // pastreaza numele fisierului

MPI_Init(&argc, &argv);        // initializarea mediului MPI


// aflam rangul procesului curent (comunicatorul este cel
   implicit)
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
// aflam dimensiunea grupului de procese lansate in executie
MPI_Comm_size(MPI_COMM_WORLD, &nProc);

strcpy(fname,"ftest1.dat");            // numele fisierului
// deschidem fisierul pentru citire de date
MPI_File_open(MPI_COMM_WORLD, fname, MPI_MODE_RDONLY,
   MPI_INFO_NULL, &fh);

// aflam dimensiunea fisierului, in octeti
MPI_File_get_size(fh, &fileSize);


// numarul local de valori float de citit, pentru fiecare
   proces
dataSize = fileSize / sizeof(float) / nProc + 1;


// alocam spatiul necesar in memoria libera, pentru cele
   dataSize valori float

data = (float *) malloc(dataSize * sizeof(float));

// pozitionam in locul corect pointerii de fisier individuali
MPI_File_set_view(fh, myRank * dataSize * sizeof(float),
   MPI_FLOAT, MPI_FLOAT, "native", MPI_INFO_NULL);

// ...si citim datele, in numar de dataSize
MPI_File_read(fh, data, dataSize, MPI_FLOAT, &status);
// aflam numarul de elemente primit de fiecare proces
MPI_Get_count(&status, MPI_FLOAT, &dataElements);

// ...si verificam rezultatul
```

```
  printf("Procesul %d a citit %d valori reale, prima fiind: %.2f\
      n", myRank, dataElements, *data);

  MPI_File_close(&fh);  // inchidem fisierul de date
  MPI_Finalize();       // terminam sesiunea MPI

  free(data);           // eliberam spatiul RAM alocat
  data = NULL;
  return 0;             // ...si iesim

}
```

2. Accesarea în paralel a unui fişier de date pentru scriere, cu utilizarea funcţiei `MPI_File_write_at()`

```
/*************************************************
 Programul demonstreaza scrierea in paralel intr-un fisier, cu
     ajutorul functiilor MPI.
*************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"


#define DATASIZE 25

int main(int argc, char **argv)

{
  int myRank;   // rangul procesului curent
  int i;

  float * data;
        // adresa de inceput a zonei RAM unde vor fi stocate
           datele
```

```
char fname[32];   // pentru numele de fisier
MPI_File fh;
      // descriptorul de fisier (handle)
MPI_Offset offset;    // offset-ul pointerului de fisier



MPI_Init(&argc, &argv);   // initializam mediul MPI

MPI_Comm_rank(MPI_COMM_WORLD, &myRank);   // aflam rangul
    fiecarui proces


strcpy(fname, "datafile.dat");   // numele fisierului

// alocam spatiu in memoria libera pentru cele DATASIZE valori
    float
data = (float *) malloc(DATASIZE * sizeof(float));
for(i = 0; i < DATASIZE; i++)
  *(data + i) = ((float)(myRank + 1))/DATASIZE + i;

// deschidem fisierul pentru scriere sau il cream, daca nu
    exista
MPI_File_open(MPI_COMM_WORLD, fname, (MPI_MODE_WRONLY |
    MPI_MODE_CREATE), MPI_INFO_NULL, &fh);

// resetam pointerii de fisier
MPI_File_set_view(fh, 0, MPI_FLOAT, MPI_FLOAT, "native",
    MPI_INFO_NULL);

// pentru fiecare proces offset-ul este definit in functie de
    rang
offset = myRank * DATASIZE;
// fiecare proces va scrie datele disponibile in acelasi fisier
    , la offset-ul corect
MPI_File_write_at(fh, offset, data, DATASIZE, MPI_FLOAT,
    MPI_STATUS_IGNORE);

MPI_File_close(&fh);  // inchidem fisierul de date

MPI_Finalize();   // terminam sesiunea MPI_Comm_rank
free(data);       // eliberam memoria alocata
```

```
    data = NULL;

    return 0;           // si iesim

}
```

Capitolul 7

Implementarea schemelor master-slave, client-server, task pool, producer-consumner

Există mai multe scheme de lucru ce pot fi alese pentru a organiza procesele în programarea paralelă [2]. Fiecare dintre ele este aplicată în acele probleme paralelizabile în care folosirea lor s-a dovedit a fi mai eficientă. Utilizatorul ar trebui sa aibă cunoştinţă de toate aceste posibilităţi şi să o aleagă pe cea pe care o consideră optimă, întrucât această alegere precum şi organizarea fluxului de date au o importanţă fundamentală în eficientizarea programelor paralele.

General vorbind, schemele principale de lucru sunt: *master-slave*, *client-server*, *task pool*, *producer-consumer*.

7.1 Master-Slave

Această schemă de lucru cunoscută şi ca *manager-worker* este printre cele mai utilizate scheme. Un proces ia rolul procesului de tip *master* (*manager*), execută părţi ale job-ului global ce nu pot fi paralelizate apoi împarte şi trimite proceselor de tip *slave* (*worker*) părţile din job ce pot fi paralelizate. Acest lucru este ilustrat în Fig. 7.1. Când un proces *worker* îşi termină sarcina trimite rezultatul obţinut înapoi la *master*. Ulterior, *master*-ul trimite alte sarcini către procesele de tip *worker*. Cel mai mare dezavantaj este acela că *master*-ul coordonează tot schimbul de informaţii, iar în unele cazuri acest lucru poate duce la un blocaj. În unele cazuri particulare se poate

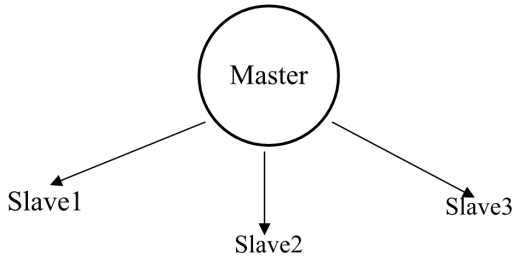

Figura 7.1: Schema master-slave

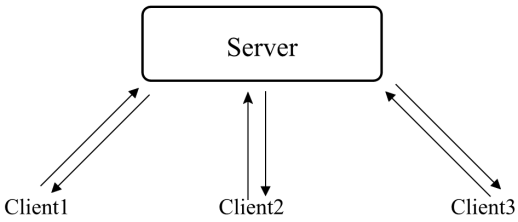

Figura 7.2: Schema client-server

întâmpla ca *master*-ul să facă şi o parte din sarcini, nu doar să aştepte rezultatele de la procesele de tip *worker*, preluând temporar rolul de *worker*.

7.2 Client-Server

În cadrul acestei scheme se generează procese multiple care trimit *request*-uri către server, apoi efectuează calcule pe datele obţinute. Schema client-server (Fig. 7.2) poate fi considerată un caz particular al schemei consumer-producer dacă vom considera că producerii sunt clienţi care practic "produc" cereri pe care le stochează într-o structură, de exemplu un buffer, iar server-ul va juca rolul de a "consuma" cererile prin accesare.

7.3 Task pool

Task pool (Fig. 7.3) reprezintă o structură de date în care datele sunt stocate şi din care pot fi extrase pentru execuţie. Principala diferenţă dintre task pool şi

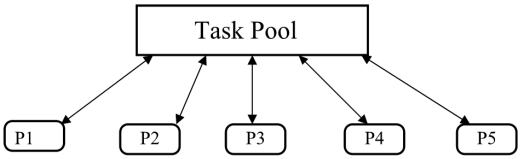

Figura 7.3: Schema Task pool

master-slave este aceea că în primul caz toate procesele sunt de acelaşi tip, neexistând un proces de tip master şi în această schemă se vor folosi un număr fix de procese ce vor fi finalizate după ce toate sarcinile au fost epuizate. În timpul realizării unei sarcini un proces poate genera noi sarcini ce vor fi introduse în task pool. Totodată, accesul la task pool trebuie sa fie sincronizat pentru a evita accesul multiplu al unor procese pe o sarcină. Structura de tip task pool poate fi pusă la dispoziţie de mediul de programare (ex. interfaţa Executor-Java) sau poate fi inclusă în codul paralel.

7.4 Producer-Consumer

După divizarea programului în sub-taskuri ce vor fi asignate unui proces, programatorul trebuie să determine modul în care aceste sub-task-uri vor fi organizate şi coordonate. Deoarece unele procesele vor cere informaţii asociate altor procese, va trebui să fim atenţi la fluxul de informaţii dintre toate procesele.

O metodă de a înţelege modul în care pot fi coordonate diferite procese care contribuie la realizarea unui job global este folosirea schemei *producer-consumer*, care este o abordare generală a modului în care două procese pot fi organizate. Cazul general este acela în care avem un set de procese ce joacă rolul producătorilor, acestea produc elemente ce sunt stocate în *buffer* sau în *work pool*. Restul proceselor vor juca rolul consumer-ului şi vor citi elementele din *buffer* sau *work pool* şi le vor folosi pentru anumite task-uri (Fig. 7.4).

Observaţie : Un producer poate scrie doar dacă buffer-ul nu este plin, iar un consumer poate accesa un buffer doar dacă acesta nu este gol; se utilizează sincronizarea proceselor.

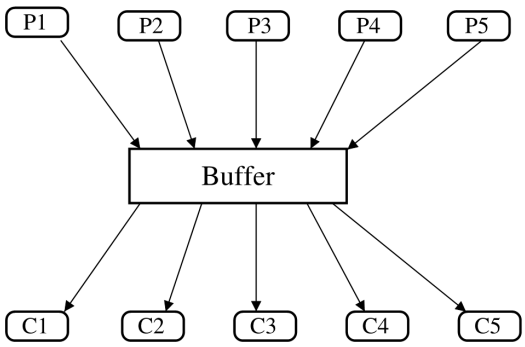

Figura 7.4: Schema producer-consumer

Referinţe

[1] Michael Kerrisk,
The Linux Programming Interface: A Linux and UNIX System Programming Handbook, No Starch Press Inc., San Francisco, 2010.

[2] Thomas Rauber and Gudula Rünger,
Parallel Programming for Multicore and Cluster Systems,
Springer-Verlag Berlin Heidelberg 2010.

[3] Peter Pacheco,
An Introduction to Parallel Programming,
Morgan Kaufmann Publishers Inc. 2011.

[4] Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill,
Patterns for Parallel Programming,
Pearson Education Inc., 2008.

[5] William Gropp, Ewing Lusk, Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*,
The MIT Press, Cambridge, Massachusetts, 2014.